

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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

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

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

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

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

Copyright © 2010-2026 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

8 March 2024

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AArch32 -- Base Instructions (alphabetic order)

[ADC, ADCS \(immediate\)](#): Add with Carry (immediate).

[ADC, ADCS \(register\)](#): Add with Carry (register).

[ADC, ADCS \(register-shifted register\)](#): Add with Carry (register-shifted register).

[ADD \(immediate, to PC\)](#): Add to PC: an alias of ADR.

[ADD, ADDS \(immediate\)](#): Add (immediate).

[ADD, ADDS \(register\)](#): Add (register).

[ADD, ADDS \(register-shifted register\)](#): Add (register-shifted register).

[ADD, ADDS \(SP plus immediate\)](#): Add to SP (immediate).

[ADD, ADDS \(SP plus register\)](#): Add to SP (register).

[ADR](#): Form PC-relative address.

[AND, ANDS \(immediate\)](#): Bitwise AND (immediate).

[AND, ANDS \(register\)](#): Bitwise AND (register).

[AND, ANDS \(register-shifted register\)](#): Bitwise AND (register-shifted register).

[ASR \(immediate\)](#): Arithmetic Shift Right (immediate): an alias of MOV, MOVS (register).

[ASR \(register\)](#): Arithmetic Shift Right (register): an alias of MOV, MOVS (register-shifted register).

[ASRS \(immediate\)](#): Arithmetic Shift Right, setting flags (immediate): an alias of MOV, MOVS (register).

[ASRS \(register\)](#): Arithmetic Shift Right, setting flags (register): an alias of MOV, MOVS (register-shifted register).

[B](#): Branch.

[BFC](#): Bit Field Clear.

[BFI](#): Bit Field Insert.

[BIC, BICS \(immediate\)](#): Bitwise Bit Clear (immediate).

[BIC, BICS \(register\)](#): Bitwise Bit Clear (register).

[BIC, BICS \(register-shifted register\)](#): Bitwise Bit Clear (register-shifted register).

[BKPT](#): Breakpoint.

[BL, BLX \(immediate\)](#): Branch with Link and optional Exchange (immediate).

[BLX \(register\)](#): Branch with Link and Exchange (register).

[BX](#): Branch and Exchange.

[BXJ](#): Branch and Exchange, previously Branch and Exchange Jazelle.

[CBNZ, CBZ](#): Compare and Branch on Nonzero or Zero.

[CLRBHB](#): Clear Branch History.

[CLREX](#): Clear-Exclusive.

[CLZ](#): Count Leading Zeros.

[CMN \(immediate\)](#): Compare Negative (immediate).

[CMN \(register\)](#): Compare Negative (register).

[CMN \(register-shifted register\)](#): Compare Negative (register-shifted register).

[CMP \(immediate\)](#): Compare (immediate).

[CMP \(register\)](#): Compare (register).

[CMP \(register-shifted register\)](#): Compare (register-shifted register).

[CPS, CPSID, CPSIE](#): Change PE State.

[CRC32](#): CRC32.

[CRC32C](#): CRC32C.

[CSDB](#): Consumption of Speculative Data Barrier.

[DBG](#): Debug hint.

[DCPS1](#): Debug Change PE State to EL1.

[DCPS2](#): Debug Change PE State to EL2.

[DCPS3](#): Debug Change PE State to EL3.

[DMB](#): Data Memory Barrier.

[DSB](#): Data Synchronization Barrier.

[EOR, EORS \(immediate\)](#): Bitwise Exclusive-OR (immediate).

[EOR, EORS \(register\)](#): Bitwise Exclusive-OR (register).

[EOR, EORS \(register-shifted register\)](#): Bitwise Exclusive-OR (register-shifted register).

[ERET](#): Exception Return.

[ESB](#): Error Synchronization Barrier.

[HLT](#): Halting Breakpoint.

[HVC](#): Hypervisor Call.

[ISB](#): Instruction Synchronization Barrier.

[IT](#): If-Then.

[LDA](#): Load-Acquire Word.

[LDAB](#): Load-Acquire Byte.

[LDAEX](#): Load-Acquire Exclusive Word.

[LDAEXB](#): Load-Acquire Exclusive Byte.

[LDAEXD](#): Load-Acquire Exclusive Doubleword.

[LDAEXH](#): Load-Acquire Exclusive Halfword.

[LDAH](#): Load-Acquire Halfword.

[LDC \(immediate\)](#): Load data to System register (immediate).

[LDC \(literal\)](#): Load data to System register (literal).

[LDM \(exception return\)](#): Load Multiple (exception return).

[LDM \(User registers\)](#): Load Multiple (User registers).

[LDM, LDMIA, LDMFD](#): Load Multiple (Increment After, Full Descending).

[LDMDA, LDMFA](#): Load Multiple Decrement After (Full Ascending).

[LDMDB, LDMEA](#): Load Multiple Decrement Before (Empty Ascending).

[LDMIB, LDMED](#): Load Multiple Increment Before (Empty Descending).

[LDR \(immediate\)](#): Load Register (immediate).

[LDR \(literal\)](#): Load Register (literal).

[LDR \(register\)](#): Load Register (register).

[LDRB \(immediate\)](#): Load Register Byte (immediate).

[LDRB \(literal\)](#): Load Register Byte (literal).

[LDRB \(register\)](#): Load Register Byte (register).

[LDRBT](#): Load Register Byte Unprivileged.

[LDRD \(immediate\)](#): Load Register Dual (immediate).

[LDRD \(literal\)](#): Load Register Dual (literal).

[LDRD \(register\)](#): Load Register Dual (register).

[LDREX](#): Load Register Exclusive.

[LDREXB](#): Load Register Exclusive Byte.

[LDREXD](#): Load Register Exclusive Doubleword.

[LDREXH](#): Load Register Exclusive Halfword.

[LDRH \(immediate\)](#): Load Register Halfword (immediate).

[LDRH \(literal\)](#): Load Register Halfword (literal).

[LDRH \(register\)](#): Load Register Halfword (register).

[LDRHT](#): Load Register Halfword Unprivileged.

[LDRSB \(immediate\)](#): Load Register Signed Byte (immediate).

[LDRSB \(literal\)](#): Load Register Signed Byte (literal).

[LDRSB \(register\)](#): Load Register Signed Byte (register).

[LDRSBT](#): Load Register Signed Byte Unprivileged.

[LDRSH \(immediate\)](#): Load Register Signed Halfword (immediate).

[LDRSH \(literal\)](#): Load Register Signed Halfword (literal).

[LDRSH \(register\)](#): Load Register Signed Halfword (register).

[LDRSHT](#): Load Register Signed Halfword Unprivileged.

[LDRT](#): Load Register Unprivileged.

[LSL \(immediate\)](#): Logical Shift Left (immediate): an alias of MOV, MOVS (register).

[LSL \(register\)](#): Logical Shift Left (register): an alias of MOV, MOVS (register-shifted register).

[LSLS \(immediate\)](#): Logical Shift Left, setting flags (immediate): an alias of MOV, MOVS (register).

[LSLS \(register\)](#): Logical Shift Left, setting flags (register): an alias of MOV, MOVS (register-shifted register).

[LSR \(immediate\)](#): Logical Shift Right (immediate): an alias of MOV, MOVS (register).

[LSR \(register\)](#): Logical Shift Right (register): an alias of MOV, MOVS (register-shifted register).

[LSRS \(immediate\)](#): Logical Shift Right, setting flags (immediate): an alias of MOV, MOVS (register).

[LSRS \(register\)](#): Logical Shift Right, setting flags (register): an alias of MOV, MOVS (register-shifted register).

[MCR](#): Move to System register from general-purpose register or execute a System instruction.

[MCRR](#): Move to System register from two general-purpose registers.

[MLA, MLAS](#): Multiply Accumulate.

[MLS](#): Multiply and Subtract.

[MOV, MOVS \(immediate\)](#): Move (immediate).

[MOV, MOVS \(register\)](#): Move (register).

[MOV, MOVS \(register-shifted register\)](#): Move (register-shifted register).

[MOVT](#): Move Top.

[MRC](#): Move to general-purpose register from System register.

[MRRC](#): Move to two general-purpose registers from System register.

[MRS](#): Move Special register to general-purpose register.

[MRS \(Banked register\)](#): Move Banked or Special register to general-purpose register.

[MSR \(Banked register\)](#): Move general-purpose register to Banked or Special register.

[MSR \(immediate\)](#): Move immediate value to Special register.

[MSR \(register\)](#): Move general-purpose register to Special register.

[MUL, Muls](#): Multiply.

[MVN, MVNS \(immediate\)](#): Bitwise NOT (immediate).

[MVN, MVNS \(register\)](#): Bitwise NOT (register).

[MVN, MVNS \(register-shifted register\)](#): Bitwise NOT (register-shifted register).

[NOP](#): No Operation.

[ORN, ORNS \(immediate\)](#): Bitwise OR NOT (immediate).

[ORN, ORNS \(register\)](#): Bitwise OR NOT (register).

[ORR, ORRS \(immediate\)](#): Bitwise OR (immediate).

[ORR, ORRS \(register\)](#): Bitwise OR (register).

[ORR, ORRS \(register-shifted register\)](#): Bitwise OR (register-shifted register).

[PKHBT, PKHTB](#): Pack Halfword.

[PLD \(literal\)](#): Preload Data (literal).

[PLD, PLDW \(immediate\)](#): Preload Data (immediate).

[PLD, PLDW \(register\)](#): Preload Data (register).

[PLI \(immediate, literal\)](#): Preload Instruction (immediate, literal).

[PLI \(register\)](#): Preload Instruction (register).

[POP](#): Pop Multiple Registers from Stack.

[POP \(multiple registers\)](#): Pop Multiple Registers from Stack: an alias of LDM, LDMIA, LDMFD.

[POP \(single register\)](#): Pop Single Register from Stack: an alias of LDR (immediate).

[PSSBB](#): Physical Speculative Store Bypass Barrier.

[PUSH](#): Push Multiple Registers to Stack.

[PUSH \(multiple registers\)](#): Push multiple registers to Stack: an alias of STMDB, STMFD.

[PUSH \(single register\)](#): Push Single Register to Stack: an alias of STR (immediate).

[QADD](#): Saturating Add.

[QADD16](#): Saturating Add 16.

[QADD8](#): Saturating Add 8.

[QASX](#): Saturating Add and Subtract with Exchange.

[QDADD](#): Saturating Double and Add.

[QDSUB](#): Saturating Double and Subtract.

[QSAX](#): Saturating Subtract and Add with Exchange.

[QSUB](#): Saturating Subtract.

[QSUB16](#): Saturating Subtract 16.

[QSUB8](#): Saturating Subtract 8.

[RBIT](#): Reverse Bits.

[REV](#): Byte-Reverse Word.

[REV16](#): Byte-Reverse Packed Halfword.

[REVSH](#): Byte-Reverse Signed Halfword.

[RFE](#), [RFEDA](#), [RFEDB](#), [RFEIA](#), [RFEIB](#): Return From Exception.

[ROR \(immediate\)](#): Rotate Right (immediate): an alias of MOV, MOVS (register).

[ROR \(register\)](#): Rotate Right (register): an alias of MOV, MOVS (register-shifted register).

[RORS \(immediate\)](#): Rotate Right, setting flags (immediate): an alias of MOV, MOVS (register).

[RORS \(register\)](#): Rotate Right, setting flags (register): an alias of MOV, MOVS (register-shifted register).

[RRX](#): Rotate Right with Extend: an alias of MOV, MOVS (register).

[RRXS](#): Rotate Right with Extend, setting flags: an alias of MOV, MOVS (register).

[RSB](#), [RSBS \(immediate\)](#): Reverse Subtract (immediate).

[RSB](#), [RSBS \(register\)](#): Reverse Subtract (register).

[RSB](#), [RSBS \(register-shifted register\)](#): Reverse Subtract (register-shifted register).

[RSC](#), [RSCS \(immediate\)](#): Reverse Subtract with Carry (immediate).

[RSC](#), [RSCS \(register\)](#): Reverse Subtract with Carry (register).

[RSC](#), [RSCS \(register-shifted register\)](#): Reverse Subtract (register-shifted register).

[SADD16](#): Signed Add 16.

[SADD8](#): Signed Add 8.

[SASX](#): Signed Add and Subtract with Exchange.

[SB](#): Speculation Barrier.

[SBC](#), [SBCS \(immediate\)](#): Subtract with Carry (immediate).

[SBC](#), [SBCS \(register\)](#): Subtract with Carry (register).

[SBC](#), [SBCS \(register-shifted register\)](#): Subtract with Carry (register-shifted register).

[SBFX](#): Signed Bit Field Extract.

[SDIV](#): Signed Divide.

[SEL](#): Select Bytes.

[SETEND](#): Set Endianness.

[SETPAN](#): Set Privileged Access Never.

[SEV](#): Send Event.

[SEVL](#): Send Event Local.

[SHADD16](#): Signed Halving Add 16.

[SHADD8](#): Signed Halving Add 8.

[SHASX](#): Signed Halving Add and Subtract with Exchange.

[SHSAX](#): Signed Halving Subtract and Add with Exchange.

[SHSUB16](#): Signed Halving Subtract 16.

[SHSUB8](#): Signed Halving Subtract 8.

[SMC](#): Secure Monitor Call.

[SMLABB](#), [SMLABT](#), [SMLATB](#), [SMLATT](#): Signed Multiply Accumulate (halfwords).

[SMLAD](#), [SMLADX](#): Signed Multiply Accumulate Dual.

[SMLAL](#), [SMLALS](#): Signed Multiply Accumulate Long.

[SMLALBB](#), [SMLALBT](#), [SMLALTB](#), [SMLALTT](#): Signed Multiply Accumulate Long (halfwords).

[SMLALD](#), [SMLALDX](#): Signed Multiply Accumulate Long Dual.

[SMLAWB, SMLAWT](#): Signed Multiply Accumulate (word by halfword).

[SMLSDD, SMLSDDX](#): Signed Multiply Subtract Dual.

[SMLSDD, SMLSDDX](#): Signed Multiply Subtract Long Dual.

[SMMLA, SMMLAR](#): Signed Most Significant Word Multiply Accumulate.

[SMMLS, SMMLSR](#): Signed Most Significant Word Multiply Subtract.

[SMMUL, SMMULR](#): Signed Most Significant Word Multiply.

[SMUAD, SMUADX](#): Signed Dual Multiply Add.

[SMULBB, SMULBT, SMULTB, SMULTT](#): Signed Multiply (halfwords).

[SMULL, SMULLS](#): Signed Multiply Long.

[SMULWB, SMULWT](#): Signed Multiply (word by halfword).

[SMUSD, SMUSDX](#): Signed Multiply Subtract Dual.

[SRS, SRSDA, SRSDDB, SRSIA, SRSIB](#): Store Return State.

[SSAT](#): Signed Saturate.

[SSAT16](#): Signed Saturate 16.

[SSAX](#): Signed Subtract and Add with Exchange.

[SSBB](#): Speculative Store Bypass Barrier.

[SSUB16](#): Signed Subtract 16.

[SSUB8](#): Signed Subtract 8.

[STC](#): Store data to System register.

[STL](#): Store-Release Word.

[STLB](#): Store-Release Byte.

[STLEX](#): Store-Release Exclusive Word.

[STLEXB](#): Store-Release Exclusive Byte.

[STLEXD](#): Store-Release Exclusive Doubleword.

[STLEXH](#): Store-Release Exclusive Halfword.

[STLH](#): Store-Release Halfword.

[STM \(User registers\)](#): Store Multiple (User registers).

[STM, STMIA, STMEA](#): Store Multiple (Increment After, Empty Ascending).

[STMDA, STMED](#): Store Multiple Decrement After (Empty Descending).

[STMDB, STMFD](#): Store Multiple Decrement Before (Full Descending).

[STMIB, STMFA](#): Store Multiple Increment Before (Full Ascending).

[STR \(immediate\)](#): Store Register (immediate).

[STR \(register\)](#): Store Register (register).

[STRB \(immediate\)](#): Store Register Byte (immediate).

[STRB \(register\)](#): Store Register Byte (register).

[STRBT](#): Store Register Byte Unprivileged.

[STRD \(immediate\)](#): Store Register Dual (immediate).

[STRD \(register\)](#): Store Register Dual (register).

[STREX](#): Store Register Exclusive.

[STREXB](#): Store Register Exclusive Byte.

[STREXD](#): Store Register Exclusive Doubleword.

[STREXH](#): Store Register Exclusive Halfword.

[STRH \(immediate\)](#): Store Register Halfword (immediate).

[STRH \(register\)](#): Store Register Halfword (register).

[STRHT](#): Store Register Halfword Unprivileged.

[STRT](#): Store Register Unprivileged.

[SUB \(immediate, from PC\)](#): Subtract from PC: an alias of ADR.

[SUB, SUBS \(immediate\)](#): Subtract (immediate).

[SUB, SUBS \(register\)](#): Subtract (register).

[SUB, SUBS \(register-shifted register\)](#): Subtract (register-shifted register).

[SUB, SUBS \(SP minus immediate\)](#): Subtract from SP (immediate).

[SUB, SUBS \(SP minus register\)](#): Subtract from SP (register).

[SVC](#): Supervisor Call.

[SXTAB](#): Signed Extend and Add Byte.

[SXTAB16](#): Signed Extend and Add Byte 16.

[SXTAH](#): Signed Extend and Add Halfword.

[SXTB](#): Signed Extend Byte.

[SXTB16](#): Signed Extend Byte 16.

[SXTH](#): Signed Extend Halfword.

[TBB, TBH](#): Table Branch Byte or Halfword.

[TEQ \(immediate\)](#): Test Equivalence (immediate).

[TEQ \(register\)](#): Test Equivalence (register).

[TEQ \(register-shifted register\)](#): Test Equivalence (register-shifted register).

[TSB](#): Trace Synchronization Barrier.

[TST \(immediate\)](#): Test (immediate).

[TST \(register\)](#): Test (register).

[TST \(register-shifted register\)](#): Test (register-shifted register).

[UADD16](#): Unsigned Add 16.

[UADD8](#): Unsigned Add 8.

[UASX](#): Unsigned Add and Subtract with Exchange.

[UBFX](#): Unsigned Bit Field Extract.

[UDF](#): Permanently Undefined.

[UDIV](#): Unsigned Divide.

[UHADD16](#): Unsigned Halving Add 16.

[UHADD8](#): Unsigned Halving Add 8.

[UHASX](#): Unsigned Halving Add and Subtract with Exchange.

[UHSAX](#): Unsigned Halving Subtract and Add with Exchange.

[UHSUB16](#): Unsigned Halving Subtract 16.

[UHSUB8](#): Unsigned Halving Subtract 8.

[UMAAL](#): Unsigned Multiply Accumulate Accumulate Long.

[UMLAL, UMLALS](#): Unsigned Multiply Accumulate Long.

[UMULL, UMULLS](#): Unsigned Multiply Long.

[UQADD16](#): Unsigned Saturating Add 16.

[UQADD8](#): Unsigned Saturating Add 8.

[UQASX](#): Unsigned Saturating Add and Subtract with Exchange.

[UQSAX](#): Unsigned Saturating Subtract and Add with Exchange.

[UQSUB16](#): Unsigned Saturating Subtract 16.

[UQSUB8](#): Unsigned Saturating Subtract 8.

[USAD8](#): Unsigned Sum of Absolute Differences.

[USADA8](#): Unsigned Sum of Absolute Differences and Accumulate.

[USAT](#): Unsigned Saturate.

[USAT16](#): Unsigned Saturate 16.

[USAX](#): Unsigned Subtract and Add with Exchange.

[USUB16](#): Unsigned Subtract 16.

[USUB8](#): Unsigned Subtract 8.

[UXTAB](#): Unsigned Extend and Add Byte.

[UXTAB16](#): Unsigned Extend and Add Byte 16.

[UXTAH](#): Unsigned Extend and Add Halfword.

[UXTB](#): Unsigned Extend Byte.

[UXTB16](#): Unsigned Extend Byte 16.

[UXTH](#): Unsigned Extend Halfword.

[WFE](#): Wait For Event.

[WFI](#): Wait For Interrupt.

[YIELD](#): Yield hint.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADC, ADCS (immediate)

Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register. If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result. The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	1	0	1	S	Rn				Rd				imm12											
cond								opc																							

Encoding for the ADC variant

Applies when (S == 0)

ADC{<c>}{<q>}{<Rd>, }<Rn>, #<const>

Encoding for the ADCS variant

Applies when (S == 1)

ADCS{<c>}{<q>}{<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	1	0	1	0	S	Rn				0	imm3				Rd				imm8							
op1																																

Encoding for the ADC variant

Applies when (S == 0)

ADC{<c>}{<q>}{<Rd>, }<Rn>, #<const>

Encoding for the ADCS variant

Applies when (S == 1)

ADCS{<c>}{<q>}{<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd>	For the "A1 ADC" and "A1 ADCS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. For the "T1 ADC" and "T1 ADCS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For the "A1 ADC" and "A1 ADCS" variants: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T1 ADC" and "T1 ADCS" variants: is the general-purpose source register, encoded in the "Rn" field.
<const>	For the "A1 ADC" and "A1 ADCS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values. For the "T1 ADC" and "T1 ADCS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, PSTATE.C);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADC, ADCS (register)

Add with Carry (register) adds a register value, the Carry flag value, and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	0	1	S	Rn				Rd				imm5				stype		0	Rm				
cond								opc																							

Encoding for the ADC, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

ADC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADC, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

ADC{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the ADCS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

ADCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADCS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

ADCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rdn		
op															

Encoding

ADC{<c>}{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())

ADCS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	1	0	S			Rn		(0)		imm3							imm2		stype				Rm

op1

Encoding for the ADC, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
ADC{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADC, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ADC{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ADC<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the ADCS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
ADCS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADCS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ADCS{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ADCS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 ADC, rotate right with extend", "A1 ADC, shift or rotate by value", "A1 ADCS, rotate right with extend", and "A1 ADCS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. For the "T2 ADC, rotate right with extend", "T2 ADC, shift or rotate by value", "T2 ADCS, rotate right with extend", and "T2 ADCS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For the "A1 ADC, rotate right with extend", "A1 ADC, shift or rotate by value", "A1 ADCS, rotate right with extend", and "A1 ADCS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T2 ADC, rotate right with extend", "T2 ADC, shift or rotate by value", "T2 ADCS, rotate right with extend", and "T2 ADCS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	For the "A1 ADC, rotate right with extend", "A1 ADC, shift or rotate by value", "A1 ADCS, rotate right with extend", and "A1 ADCS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T1", "T2 ADC, rotate right with extend", "T2 ADC, shift or rotate by value", "T2 ADCS, rotate right with extend", and "T2 ADCS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the "A1 ADC, shift or rotate by value" and "A1 ADCS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For the "T2 ADC, shift or rotate by value" and "T2 ADCS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

In T32 assembly:

- Outside an IT block, if ADCS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADCS <Rd>, <Rn> had been written.
- Inside an IT block, if ADC<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADC<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, PSTATE.C);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADC, ADCS (register-shifted register)

Add with Carry (register-shifted register) adds a register value, the Carry flag value, and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	0	1	S	Rn				Rd				Rs				0	stype		1	Rm			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

ADCS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

ADC{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<Rs>	Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcv;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

ADD (immediate, to PC)

Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This is a pseudo-instruction of [ADR](#). This means:

- The encodings in this description are named to match the encodings of [ADR](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [ADR](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	1	0	0	0	1	1	1	1	Rd				imm12											
cond				opc				S				Rn																			

Encoding for the A1 variant

ADD{<c>} {<q>} <Rd>, PC, #<const>

is equivalent to

[ADR](#){<c>} {<q>} <Rd>, <label>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd			imm8							
SP															

Encoding for the T1 variant

ADD{<c>} {<q>} <Rd>, PC, #<imm8>

is equivalent to

[ADR](#){<c>} {<q>} <Rd>, <label>

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	1	0	imm3				Rd				imm8							
o1				o2				Rn																							

Encoding for the T3 variant

ADD{<c>} {<q>} <Rd>, PC, #<imm12>

ADDW{<c>} {<q>} <Rd>, PC, #<imm12> // (UInt(Rd) < 8 && UInt(i :: imm3 :: imm8) < 256)

is equivalent to

[ADR](#){<c>} {<q>} <Rd>, <label>

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "T1" and "T3" variants: is the general-purpose destination register, encoded in the "Rd" field. For the "A1" variant: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC .
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
<const>	An immediate value. See Modified immediate constants in A32 instructions for the range of values.
<imm8>	Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.

Operation

The description of [ADR](#) gives the operational pseudocode for this instruction.

ADD, ADDS (immediate)

Add (immediate) adds an immediate value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) , [T3](#) and [T4](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	1	0	0	S	Rn				Rd				imm12											
cond				opc																											

Encoding for the ADD variant

Applies when (S == 0 && Rn != 11x1)

ADD{<c>}{<q>}{<Rd>, }<Rn>, #<const>

Encoding for the ADDS variant

Applies when (S == 1 && Rn != 1101)

ADDS{<c>}{<q>}{<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
if Rn == '1111' && S == '0' then See("ADR"); end;
if Rn == '1101' then See("ADD (SP plus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		
S															

Encoding

ADD<c>{<q>}{<Rd>, <Rn>, #<imm3> // (InITBlock())

ADDS{<q>}{<Rd>, <Rn>, #<imm3> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm3);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							
op															

Encoding

```

ADD<c>{<q>} <Rdn>, #<imm8> // (InITBlock() && UInt(imm8) < 8)

ADD<c>{<q>} {<Rdn>, }<Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)

ADDS{<q>} <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)

ADDS{<q>} {<Rdn>, }<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)

```

Decode for this encoding

```

let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm8);

```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	1	0	0	0	S	!= 1101				0	imm3				Rd				imm8							
op1												Rn																				

Encoding for the ADD variant

Applies when (S == 0)

```

ADD<c>{<q>} {<Rd>, }<Rn>, #<const>

ADD<c>.W {<Rd>, }<Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2)

```

Encoding for the ADDS variant

Applies when (S == 1 && Rd != 1111)

```

ADDS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

ADDS.W {<Rd>, }<Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2)

```

Decode for all variants of this encoding

```

if Rd == '1111' && S == '1' then See("CMN (immediate)"); end;
if Rn == '1101' then See("ADD (SP plus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;

```

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	1	0	0	0	0	0	!= 11x1				0	imm3				Rd				imm8							
o1						o2						Rn																				

Encoding

```

ADD<c>{<q>} {<Rd>, }<Rn>, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)

ADDW{<c>}{<q>} {<Rd>, }<Rn>, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

```

Decode for this encoding

```

if Rn == '1111' then See("ADR"); end;
if Rn == '1101' then See("ADD (SP plus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd>	For the "A1 ADD" and "A1 ADDS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. If the PC is used:
	- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <i>Pseudocode description of operations on the AArch32 general-purpose registers and the PC</i>. - For the ADDS variant, the instruction performs an exception return, that restores <i>PSTATE</i> from SPSR_<current_mode>. Arm deprecates use of this instruction.
	For the "T1", "T3 ADD", "T3 ADDS", and "T4" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For the "A1 ADD", "A1 ADDS", and "T4" variants: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see <i>ADD (SP plus immediate)</i> . If the PC is used, see <i>ADR</i> . For the "T1" variant: is the general-purpose source register, encoded in the "Rn" field. For the "T3 ADD" and "T3 ADDS" variants: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see <i>ADD (SP plus immediate)</i> .
<const>	For the "A1 ADD" and "A1 ADDS" variants: an immediate value. See <i>Modified immediate constants in A32 instructions</i> for the range of values. For the "T3 ADD" and "T3 ADDS" variants: an immediate value. See <i>Modified immediate constants in T32 instructions</i> for the range of values.
<imm3>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.
<Rdn>	Is the general-purpose source and destination register, encoded in the "Rdn" field.
<imm8>	Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the *ADDW* syntax). Encoding T1 is preferred to encoding T2 if <Rd> is specified and encoding T2 is preferred to encoding T1 if <Rd> is omitted.

Operation

```

if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcw) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, '0');
    if d == 15 then // Can only occur for A32 encoding
      if setflags then
        ALUExceptionReturn(result);
      else
        ALUWritePC(result);
      end;
    else
      R(d) = result;
      if setflags then
        PSTATE.[N,Z,C,V] = nzcw;
      end;
    end;
  end;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcw) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, '0');
    R(d) = result;
    if setflags then
      PSTATE.[N,Z,C,V] = nzcw;
    end;
  end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD, ADDS (register)

Add (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	0	0	S	!= 1101				Rd				imm5				stype	0	Rm					
cond				opc				Rn																							

Encoding for the ADD, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

```
ADD{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADD, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

```
ADD{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the ADDS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

```
ADDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ADDS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

```
ADDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rn == '1101' then See("ADD (SP plus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	Rm			Rn			Rd		
S															

Encoding

```
ADD<c>{<q>} <Rd>, <Rn>, <Rm> // (InITBlock())
```

```
ADDS{<q>} {<Rd>, }<Rn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	DN	!= 1101					Rdn	
op								Rm							

Encoding

Applies when $(!(DN == 1 \ \&\& \ Rdn == 101))$

ADD{<c>}{<q>}{<Rdn>, }<Rdn>, <Rm>

ADD<c>{<q>}<Rdn>, <Rm> // (Preferred syntax, Inside IT block)

Decode for this encoding

```
if (DN::Rdn) == '1101' || Rm == '1101' then See("ADD (SP plus register)"); end;
let d : integer = UInt(DN::Rdn);
let n : integer = d;
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if n == 15 && m == 15 then UnpredictableProcedure(); end;
if d == 15 && !InitBlock() && !LastInitBlock() then UnpredictableProcedure(); end;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	0	0	S	!= 1101				(0)	imm3		Rd			imm2		stype		Rm					
op1								Rn																							

Encoding for the ADD, rotate right with extend variant

Applies when $(S == 0 \ \&\& \ imm3 == 000 \ \&\& \ imm2 == 00 \ \&\& \ stype == 11)$

ADD{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADD, shift or rotate by value variant

Applies when $(S == 0 \ \&\& \ !(imm3 == 000 \ \&\& \ imm2 == 00 \ \&\& \ stype == 11))$

ADD{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

ADD<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

ADD{<c>}.W {<Rd>, }<Rn>, <Rm> // (<Rd> == <Rn>, and <Rd>, <Rn>, <Rm> can be represented in T2)

Encoding for the ADDS, rotate right with extend variant

Applies when $(S == 1 \ \&\& \ imm3 == 000 \ \&\& \ Rd != 1111 \ \&\& \ imm2 == 00 \ \&\& \ stype == 11)$

ADDS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADDS, shift or rotate by value variant

Applies when $(S == 1 \ \&\& \ Rd != 1111 \ \&\& \ !(imm3 == 000 \ \&\& \ imm2 == 00 \ \&\& \ stype == 11))$

ADDS{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

ADDS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2)

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (register)"); end;
if Rn == '1101' then See("ADD (SP plus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd>	For the "A1 ADD, rotate right with extend", "A1 ADD, shift or rotate by value", "A1 ADDS, rotate right with extend", and "A1 ADDS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. If the PC is used: - For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction. For the "T1" variant: is the general-purpose destination register, encoded in the "Rd" field. When used inside an IT block, <Rd> must be specified. When used outside an IT block, <Rd> is optional, and: - If omitted, this register is the same as <Rn>. - If present, encoding T1 is preferred to encoding T2.										
<Rn>	For the "A1 ADD, rotate right with extend", "A1 ADD, shift or rotate by value", "A1 ADDS, rotate right with extend", and "A1 ADDS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used. If the SP is used, see ADD (SP plus register). For the "T1" variant: is the first general-purpose source register, encoded in the "Rn" field. For the "T3 ADD, rotate right with extend", "T3 ADD, shift or rotate by value", "T3 ADDS, rotate right with extend", and "T3 ADDS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD (SP plus register).										
<Rm>	For the "A1 ADD, rotate right with extend", "A1 ADD, shift or rotate by value", "A1 ADDS, rotate right with extend", and "A1 ADDS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T1", "T3 ADD, rotate right with extend", "T3 ADD, shift or rotate by value", "T3 ADDS, rotate right with extend", and "T3 ADDS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. For the "T2" variant: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table> <tr> <th>stype</th><th><shift></th></tr> <tr> <td>00</td><td>LSL</td></tr> <tr> <td>01</td><td>LSR</td></tr> <tr> <td>10</td><td>ASR</td></tr> <tr> <td>11</td><td>ROR</td></tr> </table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the "A1 ADD, shift or rotate by value" and "A1 ADDS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32. For the "T3 ADD, shift or rotate by value" and "T3 ADDS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.										
<Rdn>	Is the general-purpose source and destination register, encoded in the "DN:Rdn" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. The assembler language allows <Rdn> to be specified once or twice in the assembler syntax. When used inside an IT block, and <Rdn> and <Rm> are in the range R0 to R7, <Rdn> must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when <Rdn> is specified once or twice. Inside an IT block, if ADD<c> <Rd>, <Rn>, <Rd> cannot be assembled using encoding T1, it is assembled using encoding T2 as though ADD<c> <Rd>, <Rn> had been written. To prevent this happening, use the .W qualifier.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.\[N,Z,C,V\] = nzcvc;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

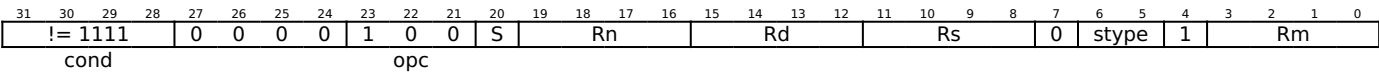
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD, ADDS (register-shifted register)

Add (register-shifted register) adds a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1



Encoding for the Flag setting variant

Applies when (S == 1)

```
ADDS{<c>}{<q>}{<Rd>}, <Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when (S == 0)

```
ADD{<c>}{<q>}{<Rd>}, <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<Rs>	Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcv;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

ADD, ADDS (SP plus immediate)

Add to SP (immediate) adds an immediate value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) , [T3](#) and [T4](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	1	0	1	0	0	S	1	1	0	1	Rd					imm12											
cond				opc								Rn																				

Encoding for the ADD variant

Applies when (S == 0)

```
ADD{<c>}{<q>} {<Rd>, }SP, #<const>
```

Encoding for the ADDS variant

Applies when (S == 1)

```
ADDS{<c>}{<q>} {<Rd>, }SP, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	Rd			imm8							
SP															

Encoding

```
ADD{<c>}{<q>} <Rd>, SP, #<imm8>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	imm7						
S															

Encoding

```
ADD{<c>}{<q>} {SP, }SP, #<imm7>
```

Decode for this encoding

```
let d : integer = 13;
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm7::'00');
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	i	0	1	0	0	0	S	1	1	0	1	0	imm3			Rd				imm8										
op1											Rn																							

Encoding for the ADD variant

Applies when (S == 0)

```
ADD{<c>}{<q>} {<Rd>, }SP, #<const>
```

```
ADD{<c>}.W {<Rd>, }SP, #<const> // (<Rd>, <const> can be represented in T1 or T2)
```

Encoding for the ADDS variant

Applies when (S == 1 && Rd != 1111)

```
ADDS{<c>}{<q>} {<Rd>, }SP, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (immediate)"); end;
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if d == 15 && !setflags then UnpredictableProcedure(); end;
```

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	1	0	0	0	0	0	1	1	0	1	0	imm3				Rd				imm8							
								o1				o2				Rn																

Encoding

```
ADD{<c>}{<q>} {<Rd>, }SP, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)
```

```
ADDW{<c>}{<q>} {<Rd>, }SP, #<imm12> // (<imm12> can be represented in T1, T2, or T3)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 ADD" and "A1 ADDS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used: - For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
<const>	For the "A1 ADD" and "A1 ADDS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values. For the "T3 ADD" and "T3 ADDS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.
<imm8>	Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.
SP	Is the stack pointer.
<imm7>	Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), imm32, '0');
  if d == 15 then // Can only occur for A32 encoding
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
    end;
  else
    R(d) = result;
    if setflags then
      PSTATE.[N,Z,C,V] = nzcvc;
    end;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD, ADDS (SP plus register)

Add to SP (register) adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADDS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	0	0	S	1	1	0	1	Rd				imm5				stype		0	Rm				
cond				opc				Rn																							

Encoding for the ADD, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

```
ADD{<c>}{<q>} {<Rd>, }SP, <Rm> , RRX
```

Encoding for the ADD, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

```
ADD{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Encoding for the ADDS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

```
ADDS{<c>}{<q>} {<Rd>, }SP, <Rm> , RRX
```

Encoding for the ADDS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

```
ADDS{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	DM	1	1	0	1	Rdm		
op									Rs						

Encoding

```
ADD{<c>}{<q>} {<Rdm>, }SP, <Rdm>
```

Decode for this encoding

```
let d : integer = UInt(DM::Rdm);
let m : integer = UInt(DM::Rdm);
let setflags : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if d == 15 && InitTBlock() && !LastInTBlock() then UnpredictableProcedure(); end;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	!= 1101				1	0	1
op								D	Rm				Rd		

Encoding

ADD{<c>}{<q>} {SP, }SP, <Rm>

Decode for this encoding

```
if Rm == '1101' then See("encoding T1"); end;
let d : integer = 13;
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	0	0	S	1	1	0	1	(0)	imm3							imm2	stype						
op1							Rn														Rd			imm2	stype	Rm					

Encoding for the ADD, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

ADD{<c>}{<q>} {<Rd>, }SP, <Rm>, RRX

Encoding for the ADD, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ADD{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}

ADD{<c>}.W {<Rd>, }SP, <Rm> // (<Rd>, <Rm> can be represented in T1 or T2)

Encoding for the ADDS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

ADDS{<c>}{<q>} {<Rd>, }SP, <Rm>, RRX

Encoding for the ADDS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ADDS{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (register)"); end;
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> For the "A1 ADD, rotate right with extend", "A1 ADD, shift or rotate by value", "A1 ADDS, rotate right with extend", and "A1 ADDS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ADDS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the "T3 ADD, rotate right with extend", "T3 ADD, shift or rotate by value", "T3 ADDS, rotate right with extend", and "T3 ADDS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<Rm> For the "A1 ADD, rotate right with extend", "A1 ADD, shift or rotate by value", "A1 ADDS, rotate right with extend", "A1 ADDS, shift or rotate by value", and "T2" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T3 ADD, rotate right with extend", "T3 ADD, shift or rotate by value", "T3 ADDS, rotate right with extend", and "T3 ADDS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in "stype":

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the "A1 ADD, shift or rotate by value" and "A1 ADDS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For the "T3 ADD, shift or rotate by value" and "T3 ADDS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

<Rdm> Is the general-purpose destination and second source register, encoded in the "Rdm" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

SP Is the stack pointer.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), shifted, '0');
    if d == 15 then
        if setflags then
            ALUEXCEPTIONReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

This instruction is used by the alias [SUB \(immediate, from PC\)](#).

This instruction is used by the pseudo-instruction [ADD \(immediate, to PC\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	1	0	0	0	1	1	1	1	Rd				imm12											
cond				opc				S	Rn																						

Encoding

ADR{<c>}{<q>} <Rd>, <label>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = A32ExpandImm(imm12);
let add : boolean = TRUE;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	0	1	0	0	1	1	1	1	Rd				imm12											
cond				opc				S	Rn																						

Encoding

ADR{<c>}{<q>} <Rd>, <label>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = A32ExpandImm(imm12);
let add : boolean = FALSE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd			imm8							
SP															

Encoding

ADR{<c>}{<q>} <Rd>, <label>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let add : boolean = TRUE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	1	0	1	0	1	0	1	1	1	1	0	imm3				Rd				imm8							
				o1				o2				Rn																				

Encoding

ADR{<c>}{<q>} <Rd>, <label>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = ZeroExtend{(i::imm3::imm8)};
let add : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	i	1	0	0	0	0	0	1	1	1	1	0	imm3				Rd				imm8								
						o1				o2				Rn																			

Encoding

```
ADR{<c>}{<q>}<Rd>, <label>

ADR{<c>}.W <Rd>, <label> // (<Rd>, <label> can be presented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = ZeroExtend{(i::imm3::imm8)};
let add : boolean = TRUE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> For the "A1" and "A2" variants: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
For the "T1", "T2", and "T3" variants: is the general-purpose destination register, encoded in the "Rd" field.
- <label> For the "A1" and "A2" variants: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset. If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32. Permitted values of the size of the offset are any of the constants described in [Modified immediate constants in A32 instructions](#).
For the "T1" variant: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.
For the "T2" and "T3" variants: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset. If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32. Permitted values of the size of the offset are 0-4095.

The instruction aliases permit the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Alias Conditions

Alias	Of variant	Is preferred when
ADD (immediate, to PC)		Never
SUB (immediate, from PC)	A2	<code>imm12 == '000000000000'</code>
SUB (immediate, from PC)	T2	<code>i :: imm3 :: imm8 == '000000000000'</code>

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let result : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                           else (AlignDownSize(PC32(),4) - imm32));
  if d == 15 then           // Can only occur for A32 encodings
    ALUWritePC(result);
  else
    R(d) = result;
  end;
end;
```

AND, ANDS (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	0	0	0	S	Rn				Rd				imm12											
cond				opc																											

Encoding for the AND variant

Applies when (S == 0)

```
AND{<c>}{<q>}{<Rd>, }<Rn>, #<const>
```

Encoding for the ANDS variant

Applies when (S == 1)

```
ANDS{<c>}{<q>}{<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	0	0	0	0	S	Rn				0	imm3				Rd				imm8							
op1																																

Encoding for the AND variant

Applies when (S == 0)

```
AND{<c>}{<q>}{<Rd>, }<Rn>, #<const>
```

Encoding for the ANDS variant

Applies when (S == 1 && Rd != 1111)

```
ANDS{<c>}{<q>}{<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TST (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd>	For the "A1 AND" and "A1 ANDS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. For the "T1 AND" and "T1 ANDS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For the "A1 AND" and "A1 ANDS" variants: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T1 AND" and "T1 ANDS" variants: is the general-purpose source register, encoded in the "Rn" field.
<const>	For the "A1 AND" and "A1 ANDS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values. For the "T1 AND" and "T1 ANDS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                            else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) AND imm32;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXCEPTIONRETURN(result);
        else
            ALUWRITEPC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND, ANDS (register)

Bitwise AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	0	0	S	Rn				Rd				imm5				stype		0	Rm				
cond								opc																							

Encoding for the AND, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

AND{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the AND, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

AND{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the ANDS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

ANDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ANDS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

ANDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		
op															

Encoding

AND{<c>}{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())

ANDS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	0	0	S			Rn		(0)		imm3			Rd			imm2		stype			Rm		
op1																															

op1

Encoding for the AND, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

AND{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the AND, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

AND{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

AND<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the ANDS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

ANDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ANDS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ANDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

ANDS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TST (register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> For the "A1 AND, rotate right with extend", "A1 AND, shift or rotate by value", "A1 ANDS, rotate right with extend", and "A1 ANDS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ANDS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the "T2 AND, rotate right with extend", "T2 AND, shift or rotate by value", "T2 ANDS, rotate right with extend", and "T2 ANDS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For the "A1 AND, rotate right with extend", "A1 AND, shift or rotate by value", "A1 ANDS, rotate right with extend", and "A1 ANDS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For the "T2 AND, rotate right with extend", "T2 AND, shift or rotate by value", "T2 ANDS, rotate right with extend", and "T2 ANDS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For the "A1 AND, rotate right with extend", "A1 AND, shift or rotate by value", "A1 ANDS, rotate right with extend", and "A1 ANDS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T1", "T2 AND, rotate right with extend", "T2 AND, shift or rotate by value", "T2 ANDS, rotate right with extend", and "T2 ANDS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the "A1 AND, shift or rotate by value" and "A1 ANDS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For the "T2 AND, shift or rotate by value" and "T2 ANDS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

In T32 assembly:

- Outside an IT block, if ANDS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ANDS <Rd>, <Rn> had been written.
- Inside an IT block, if AND<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though AND<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C(32){R(m), shift_t, shift_n, PSTATE.C};
    let result : bits(32) = R(n) AND shifted;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit(32)(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND, ANDS (register-shifted register)

Bitwise AND (register-shifted register) performs a bitwise AND of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	0	0	S	Rn				Rd				Rs				0	stype		1	Rm			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

ANDS{<c>}{<q>}{<Rd>, <Rn>, <Rm>, <shift> <Rs>}

Encoding for the Not flag setting variant

Applies when (S == 0)

AND{<c>}{<q>}{<Rd>, <Rn>, <Rm>, <shift> <Rs>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<Rs>	Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd					imm5					1	0	0	Rm		
cond				opc				S				Rn																stype			

Encoding for the MOV, shift or rotate by value variant

ASR{<c>} {<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV{<c>} {<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	imm5					Rm			Rd		
op															

Encoding for the T2 variant

ASR<c> {<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV<c> {<q>} <Rd>, <Rm>, ASR #<imm>

and is the preferred disassembly when `InITBlock()`.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	0	1	1	1	(0)	imm3				Rd				imm2		1	0	Rm			
op1						S						Rn						stype													

Encoding for the MOV, shift or rotate by value variant

ASR{<c>} {<q>} {<Rd>, }<Rm>, #<imm>

ASR<c>.W {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)

is equivalent to

MOV{<c>} {<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 MOV, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC . For the "T2" and "T3 MOV, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1 MOV, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T2" and "T3 MOV, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.

<imm>

For the "A1 MOV, shift or rotate by value" and "T2" variants: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For the "T3 MOV, shift or rotate by value" variant: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd				Rs				0	1	0	1	Rm			
cond				opc				S				Rn												stype							

Encoding for the Not flag setting variant

ASR{<c>}{<q>}{<Rd>, }<Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>}<Rd>, <Rm>, ASR <Rs>

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0	Rs			Rdm		
op															

Encoding for the Arithmetic shift right variant

ASR<c>{<q>}{<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOV<c>{<q>}<Rdm>, <Rdm>, ASR <Rs>

and is the preferred disassembly when `InITBlock()`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	0	0	1	0	0	Rm				1	1	1	1	Rd				0	0	0	0	Rs							
stype												S																							

Encoding for the Not flag setting variant

ASR{<c>}{<q>}{<Rd>, }<Rm>, <Rs>

ASR<c>.W {<Rd>, }<Rm>, <Rs> // (InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))

is equivalent to

MOV{<c>}{<q>}<Rd>, <Rm>, ASR <Rs>

and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASRS (immediate)

Arithmetic Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd				imm5					1	0	0	Rm			
cond				opc				S	Rn												stypc										

Encoding for the MOVS, shift or rotate by value variant

ASRS{<c>}{<q>}{<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>}{<Rd>, }<Rm>, ASR #<imm>

and is always the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	imm5					Rm			Rd		
op															

Encoding for the T2 variant

ASRS{<q>}{<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS<c>{<q>}{<Rd>, }<Rm>, ASR #<imm>

and is the preferred disassembly when `!InITBlock()`.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																								
1			1			1			0			1			0			1			0			1			1			1			1			(0)			imm3			Rd				imm2		1		0		Rm			
op1												S				Rn																				stypc																			

Encoding for the MOVS, shift or rotate by value variant

ASRS{<c>}{<q>}{<Rd>, }<Rm>, #<imm>

ASRS.W {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)

is equivalent to

MOVS{<c>}{<q>}{<Rd>, }<Rm>, ASR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd>	For the "A1 MOVS, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores <i>PSTATE</i> from SPSR_<current_mode>. For the "T2" and "T3 MOVS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1 MOVS, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T2" and "T3 MOVS, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For the "A1 MOVS, shift or rotate by value" and "T2" variants: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32. For the "T3 MOVS, shift or rotate by value" variant: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASRS (register)

Arithmetic Shift Right, setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd				Rs				0	1	0	1	Rm			
cond								opc				S				Rn								stype							

Encoding for the Flag setting variant

ASRS{<c>}{<q>}{<Rd>, <Rm>, <Rs>}

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, ASR <Rs>

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0	Rs			Rdm		
op															

Encoding for the Arithmetic shift right variant

ASRS{<q>}{<Rdm>, <Rdm>, <Rs>}

is equivalent to

MOVS{<q>}<Rdm>, <Rdm>, ASR <Rs>

and is the preferred disassembly when `!InITBlock()`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	1	Rm				1	1	1	1	Rd				0	0	0	0	Rs			
styp																S															

Encoding for the Flag setting variant

ASRS{<c>}{<q>}{<Rd>, <Rm>, <Rs>}

ASRS.W {<Rd>, <Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, ASR <Rs>

and is always the preferred disassembly.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

B

Branch causes a branch to a target address.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) , [T3](#) and [T4](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	0	1	0	imm24																							
cond				H																											

Encoding

B{<c>}{<q>} <label>

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{(imm24::'00')}
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 1 0 1				!= 111x				imm8							
cond															

Encoding

B<c>{<q>} <label> // (Not permitted in IT block)

Decode for this encoding

```
if cond == '1110' then See("UDF"); end;
if cond == '1111' then See("SVC"); end;
let imm32 : bits(32) = SignExtend{(imm8::'0')};
if InITBlock() then UnpredictableProcedure(); end;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	imm11										
op0			op1												

Encoding

B{<c>}{<q>} <label> // (Outside or last in IT block)

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{(imm11::'0')};
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	!= 111x					imm6					1	0	J1	0	J2	imm11										
cond																															

Encoding

B<c>.W <label> // (Not permitted in IT block, and <label> can be represented in T1)

B<c>{<q>} <label> // (Not permitted in IT block)

Decode for this encoding

```
if cond[3:1] == '111' then See("Related encodings"); end;
let imm32 : bits(32) = SignExtend{(S::J2::J1::imm6::imm11::'0')};
if InITBlock() then UnpredictableProcedure(); end;
```

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	0	J1	1	J2	imm11										

Encoding

B{<c>}{<q>}<label>

B{<c>}.W <label> // (<label> can be represented in T2)

Decode for this encoding

```
let I1 : bit = NOT(J1 XOR S);
let I2 : bit = NOT(J2 XOR S);
let imm32 : bits(32) = SignExtend{(S::I1::I2::imm10::imm11::'0')};
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Branches and miscellaneous control](#).

Assembler Symbols

<c>	For the "A1" variant: see Standard assembler syntax fields. For the "T1" and "T3" variants: see Standard assembler syntax fields. Must not be AL or omitted. For the "T2" and "T4" variants: see Standard assembler syntax fields.
<q>	See Standard assembler syntax fields.
<label>	For the "A1" variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are multiples of 4 in the range –33554432 to 33554428. For the "T1" variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –256 to 254. For the "T2" variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –2048 to 2046. For the "T3" variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –1048576 to 1048574. For the "T4" variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC32() + imm32, BranchType_DIR);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	1	1	1	1	1	0	msb				Rd				lsb				0	0	1	1	1	1	Rn			
cond																																

Encoding

BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>

Decode for this encoding

```
let d : integer = UInt(Rd);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(lsb);
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	1	1	1	0	imm3				Rd				imm2		(0)	msb				
op1												Rn																			

Encoding

BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>

Decode for this encoding

```
let d : integer = UInt(Rd);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<lsb>	For the "A1" variant: is the least significant bit to be cleared, in the range 0 to 31, encoded in the "lsb" field. For the "T1" variant: is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the "imm3:imm2" field.
<width>	Is the number of bits to be cleared, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[msbit:lsbit] = Replicate{(msbit-lsbit)+1}('0');
    // Other bits of R[d] are unchanged
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	1	0	msb				Rd				lsb				0	0	1	!= 1111					
cond				Rn																											

Encoding

BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```
if Rn == '1111' then See("BFC"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(lsb);
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	!= 1111				0	imm3				Rd				imm2		(0)	msb			
op1												Rn																			

Encoding

BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```
if Rn == '1111' then See("BFC"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the "A1" variant: is the least significant destination bit, in the range 0 to 31, encoded in the "lsb" field. For the "T1" variant: is the least significant destination bit, in the range 0 to 31, encoded in the "imm3:imm2" field.
<width>	Is the number of bits to be copied, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[msbit:lsbit] = R(n)[(msbit-lsbit):0];
    // Other bits of R[d] are unchanged
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC, BICS (immediate)

Bitwise Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	1	1	0	S	Rn				Rd				imm12											
cond										opc																					

Encoding for the BIC variant

Applies when (S == 0)

BIC{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the BICS variant

Applies when (S == 1)

BICS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	0	0	0	1	S	Rn				0	imm3				Rd				imm8							
op1																																

Encoding for the BIC variant

Applies when (S == 0)

BIC{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the BICS variant

Applies when (S == 1)

BICS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

<Rd>	For the "A1 BIC" and "A1 BICS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the BICS variant, the instruction performs an exception return, that restores <i>PSTATE</i> from SPSR_<current_mode>. For the "T1 BIC" and "T1 BICS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For the "A1 BIC" and "A1 BICS" variants: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T1 BIC" and "T1 BICS" variants: is the general-purpose source register, encoded in the "Rn" field.
<const>	For the "A1 BIC" and "A1 BICS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values. For the "T1 BIC" and "T1 BICS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                            else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) AND NOT(imm32);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC, BICS (register)

Bitwise Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	0	S	Rn				Rd				imm5					stype	0	Rm				
cond									opc																						

Encoding for the BIC, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

BIC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the BIC, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

BIC{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the BICS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

BICS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the BICS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

BICS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	0	Rm			Rdn		
op															

Encoding

BIC{<c>}{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())

BICS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	0	1	S			Rn		(0)			imm3				Rd			imm2		stype			Rm
op1																															

Encoding for the BIC, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
BIC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the BIC, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
BIC{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
BIC<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the BICS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
BICS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the BICS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
BICS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
BICS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 BIC, rotate right with extend", "A1 BIC, shift or rotate by value", "A1 BICS, rotate right with extend", and "A1 BICS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
<Rn>	For the "T2 BIC, rotate right with extend", "T2 BIC, shift or rotate by value", "T2 BICS, rotate right with extend", and "T2 BICS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. For the "A1 BIC, rotate right with extend", "A1 BIC, shift or rotate by value", "A1 BICS, rotate right with extend", and "A1 BICS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T2 BIC, rotate right with extend", "T2 BIC, shift or rotate by value", "T2 BICS, rotate right with extend", and "T2 BICS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	For the "A1 BIC, rotate right with extend", "A1 BIC, shift or rotate by value", "A1 BICS, rotate right with extend", and "A1 BICS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T1", "T2 BIC, rotate right with extend", "T2 BIC, shift or rotate by value", "T2 BICS, rotate right with extend", and "T2 BICS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.

<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”:										
	<table> <tr> <th>stype</th><th><shift></th></tr> <tr> <td>00</td><td>LSL</td></tr> <tr> <td>01</td><td>LSR</td></tr> <tr> <td>10</td><td>ASR</td></tr> <tr> <td>11</td><td>ROR</td></tr> </table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the "A1 BIC, shift or rotate by value" and "A1 BICS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32. For the "T2 BIC, shift or rotate by value" and "T2 BICS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.										
<Rdn>	Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.										

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift\_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND NOT(shifted);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

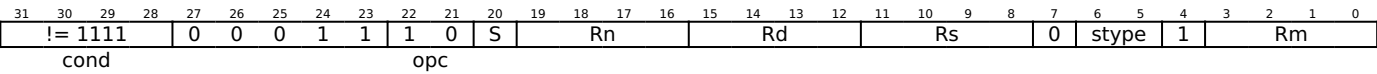
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC, BICS (register-shifted register)

Bitwise Bit Clear (register-shifted register) performs a bitwise AND of a register value and the complement of a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1



Encoding for the Flag setting variant

Applies when (S == 1)

```
BICS{<c>}{<q>}{<Rd>, <Rn>, <Rm>, <shift> <Rs>}
```

Encoding for the Not flag setting variant

Applies when (S == 0)

```
BIC{<c>}{<q>}{<Rd>, <Rn>, <Rm>, <shift> <Rs>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the type of shift to be applied to the second source register, encoded in "stype":

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND NOT(shifted);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

BKPT

Breakpoint causes a Breakpoint Instruction exception.
Breakpoint is always unconditional, even when inside an IT block.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0	1	0	imm12											0	1	1	1	imm4				
cond									opc																						

Encoding

BKPT{<q>} {#}<imm>

Decode for this encoding

```
let imm16 : bits(16) = imm12::imm4;
if cond != '1110' then UnpredictableProcedure(); end; // BKPT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	imm8							

Encoding

BKPT{<q>} {#}<imm>

Decode for this encoding

```
let imm16 : bits(16) = ZeroExtend{}(imm8);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

`<q>`
`<imm>`

See [Standard assembler syntax fields](#). A BKPT instruction must be unconditional.

For the "A1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Is ignored otherwise.

For the "T1" variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Is ignored otherwise.

Operation

```
EncodingSpecificOperations();
AArch32_SoftwareBreakpoint(imm16);
```


BL, BLX (immediate)

Branch with Link calls a subroutine at a PC-relative address, and setting LR to the return address.

Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, setting LR to the return address, and changes the instruction set from A32 to T32, or from T32 to A32.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=	1	1	1	1	1	0	1	1	imm24																						
cond								H																							

Encoding

BL{<c>}{<q>} <label>

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{(imm24::'00')};
let targetInstrSet : InstrSet = InstrSet_A32;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	H	imm24																							

Encoding

BLX{<c>}{<q>} <label>

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{(imm24::H::'0')};
let targetInstrSet : InstrSet = InstrSet_T32;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	1	J1	1	J2	imm11										

Encoding

BL{<c>}{<q>} <label>

Decode for this encoding

```
let I1 : bit = NOT(J1 XOR S);
let I2 : bit = NOT(J2 XOR S);
let imm32 : bits(32) = SignExtend{(S::I1::I2::imm10::imm11::'0')};
let targetInstrSet : InstrSet = InstrSet_T32;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10H										1	1	J1	0	J2	imm10L										H

Encoding

BLX{<c>}{<q>} <label>

Decode for this encoding

```
if H == '1' then Undefined(); end;
let I1 : bit = NOT(J1 XOR S);
let I2 : bit = NOT(J2 XOR S);
let imm32 : bits(32) = SignExtend{(S::I1::I2::imm10H::imm10L::'00')};
let targetInstrSet : InstrSet = InstrSet_A32;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> For the "A1" variant: see *Standard assembler syntax fields*.
For the "A2" variant: see *Standard assembler syntax fields*. <c> must be AL or omitted.

For the "T1" and "T2" variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<label>

For the "A1" variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are multiples of 4 in the range –33554432 to 33554428.

For the "A2" variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –33554432 to 33554430.

For the "T1" variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214.

For the "T2" variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the Align(PC, 4) value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are multiples of 4 in the range –16777216 to 16777212.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_A32 then
        LR() = PC32() - 4;
    else
        LR() = PC32()[31:1] :: '1';
    end;
    var targetAddress : bits(32);
    if targetInstrSet == InstrSet_A32 then
        targetAddress = AlignDownSize(PC32(), 4) + imm32;
    else
        targetAddress = PC32() + imm32;
    end;
    SelectInstrSet(targetInstrSet);
    BranchWritePC(targetAddress, BranchType_DIRCALL);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address specified in the register, and if necessary changes to the instruction set indicated by bit[0] of the register value. If the value in bit[0] is 0, the instruction set after the branch will be A32. If the value in bit[0] is 1, the instruction set after the branch will be T32.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	1	Rm			
cond										op0																					

Encoding

BLX{<c>} {<q>} <Rm>

Decode for this encoding

```
let m : integer = UInt(Rm);
if m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	1	Rm				(0)	(0)	(0)
L															

Encoding

BLX{<c>} {<q>} <Rm>

Decode for this encoding

```
let m : integer = UInt(Rm);
if m == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let target : bits(32) = R(m);
    var next_instr_addr : bits(32);
    if CurrentInstrSet() == InstrSet_A32 then
        next_instr_addr = PC32() - 4;
        LR() = next_instr_addr;
    else
        next_instr_addr = PC32() - 2;
        LR() = next_instr_addr[31:1] :: '1';
    end;
    BXWritePC(target, BranchType_INDCALL);
end;
```

BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0 1		0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	0	1	Rm			
cond										op0																					

Encoding

BX{<c>}{<q>} <Rm>

Decode for this encoding

```
let m : integer = UInt(Rm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0	Rm				(0)	(0)	(0)
L															

Encoding

BX{<c>}{<q>} <Rm>

Decode for this encoding

```
let m : integer = UInt(Rm);
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rm> For the "A1" variant: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used.
For the "T1" variant: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used.

Note

If <Rm> is the PC at a non word-aligned address, it results in UNPREDICTABLE behavior because the address passed to the BXWritePC() pseudocode function has bits<1:0> = '10'.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R(m), BranchType_INDIR);
end;
```

BXJ

Branch and Exchange, previously Branch and Exchange Jazelle.
BXJ behaves as a BX instruction, see [BX](#). This means it causes a branch to an address and instruction set specified by a register.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	0	Rm			
cond										op0																					

Encoding

BXJ{<c>}{<q>} <Rm>

Decode for this encoding

```
let m : integer = UInt(Rm);
if m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	0	Rm				1	0	(0)	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Encoding

BXJ{<c>}{<q>} <Rm>

Decode for this encoding

```
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R(m), BranchType_INDIR);
end;
```

CBNZ, CBZ

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	op	0	i	1	imm5					Rn		

Encoding for the CBNZ variant

Applies when (op == 1)

CBNZ{<q>} <Rn>, <label>

Encoding for the CBZ variant

Applies when (op == 0)

CBZ{<q>} <Rn>, <label>

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(i::imm5::'0')};
let nonzero : boolean = (op == '1');
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <Rn> Is the general-purpose register to be tested, encoded in the "Rn" field.
- <label> Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 and in the range 0 to 126, is encoded as "i:imm5" times 2.

Operation

```
EncodingSpecificOperations();
if nonzero != IsZero(R(n)) then
    CBWritePC(PC32() + imm32);
end;
```

CLRBHB

Clear Branch History clears the branch history for the current context to the extent that branch history information created before the CLRBHB instruction cannot be used by code before the CLRBHB instruction to exploitatively control the execution of any indirect branches in code in the current context that appear in program order after the instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1 (FEAT_CLRBHB)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=	1	1	1	1	0	0	1	1	0	0	1	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	1	1	0
cond				R				imm4								imm12															

Encoding

CLRBHB{<c>}{<q>}

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CLRBHB) then ExecuteAsNOP\(\); end;
```

T1 (FEAT_CLRBHB)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	1	1	0
												hint						option													

Encoding

CLRBHB{<c>}{<q>}

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CLRBHB) then ExecuteAsNOP\(\); end;
```

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    Hint_CLRBHB();
end;
```

CLREX

Clear-Exclusive clears the local monitor of the executing PE.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	(1)	(1)	(1)	(1)
opcode																										option					

Encoding

CLREX{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	1	0	(1)	(1)	(1)	(1)
opc																										option					

Encoding

CLREX{<c>}{<q>}

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c>
- For the "A1" variant: see [Standard assembler syntax fields](#). Must be AL or omitted.
- For the "T1" variant: see [Standard assembler syntax fields](#).
- <q>
- See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
end;
```


CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	1	0	(1)	(1)	(1)	(1)	Rd				(1)	(1)	(1)	(1)	0	0	0	1	Rm			
cond									op0																						

Encoding

CLZ{<c>}{<q>}<Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	1	Rn				1	1	1	1	Rd				1	0	0	0	Rm			
op1																op2															

Encoding

CLZ{<c>}{<q>}<Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes with the additional decode: `m = UInt(Rn)`;
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1" variant: is the general-purpose source register, encoded in the "Rm" field. For the "T1" variant: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = CountLeadingZeroBits(R(m));
    R(d) = result[31:0];
end;
```

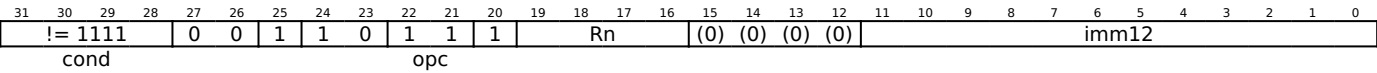
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

CMN (immediate)

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



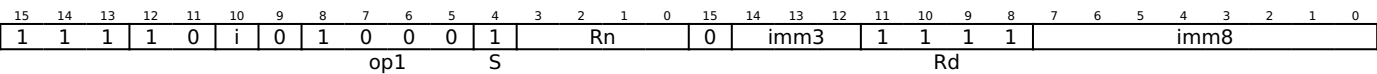
Encoding

CMN{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1



Encoding

CMN{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> For the "A1" variant: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For the "T1" variant: is the general-purpose source register, encoded in the "Rn" field.
- <const> For the "A1" variant: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.
For the "T1" variant: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, '0');
    PSTATE.[N,Z,C,V] = nzcvc;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	1	1	Rn				(0)	(0)	(0)	(0)	imm5				stype		0	Rm				
cond									opc																						

Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

CMN{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

Applies when (!(imm5 == 00000 && stype == 11))

CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	1	Rm			Rn		
op															

Encoding

CMN{<c>}{<q>} <Rn>, <Rm>

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTType = SRTType_LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	0	0	1	Rn				(0)	imm3				1	1	1	1	imm2		stype	Rm			
op1												S	Rd																		

Encoding for the Rotate right with extend variant

Applies when (imm3 == 000 && imm2 == 00 && stype == 11)

CMN{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

Applies when (!(imm3 == 000 && imm2 == 00 && stype == 11))

CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

CMN{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q>	See <i>Standard assembler syntax fields</i> .										
<Rn>	For the "A1 Rotate right with extend" and "A1 Shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T1", "T2 Rotate right with extend", and "T2 Shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	For the "A1 Rotate right with extend" and "A1 Shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T1", "T2 Rotate right with extend", and "T2 Shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table> <tr> <th>stype</th><th><shift></th></tr> <tr> <td>00</td><td>LSL</td></tr> <tr> <td>01</td><td>LSR</td></tr> <tr> <td>10</td><td>ASR</td></tr> <tr> <td>11</td><td>ROR</td></tr> </table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the "A1 Shift or rotate by value" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32. For the "T2 Shift or rotate by value" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.										

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    PSTATE.[N,Z,C,V] = nzcv;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

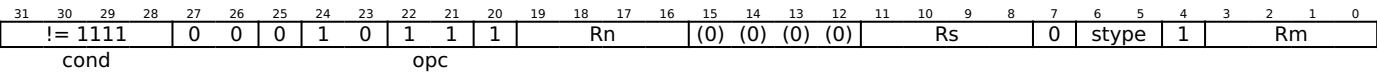
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMN (register-shifted register)

Compare Negative (register-shifted register) adds a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1



Encoding

CMN{<c>} {<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SRTYPE = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <type> Is the type of shift to be applied to the second source register, encoded in "stype":

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR
- <Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let shift_n : integer = UInt(R(s)[7:0]);
  let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
  let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
  PSTATE.[N,Z,C,V] = nzcvc;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	1	0	1	Rn				(0)	(0)	(0)	(0)	imm12											
cond												opc																			

Encoding

CMP{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rn			imm8							
op															

Encoding

CMP{<c>}{<q>} <Rn>, #<imm8>

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	1	1	0	1	1	Rn				0	imm3				1	1	1	1	imm8							
op1								S								Rd																

Encoding

CMP{<c>}{<q>} <Rn>, #<const>

CMP{<c>}.W <Rn>, #<const> // (<Rd>, <const> can be represented in T1)

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> For the "A1" variant: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For the "T1" variant: is a general-purpose source register, encoded in the "Rn" field.
For the "T2" variant: is the general-purpose source register, encoded in the "Rn" field.
- <const> For the "A1" variant: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.
For the "T2" variant: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.
- <imm8> Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(imm32), '1');
  PSTATE.[N,Z,C,V] = nzcvc;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	0	1	Rn				(0)	(0)	(0)	(0)	imm5					stype	0	Rm				
cond										opc																					

Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when (!(imm5 == 00000 && stype == 11))

```
CMP{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm			Rn		
op															

Encoding

```
CMP{<c>}{<q>} <Rn>, <Rm> // (<Rn> and <Rm> both from R0-R7)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTType = SRTType_LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	N	Rm			Rn			
op															

Encoding

```
CMP{<c>}{<q>} <Rn>, <Rm> // (<Rn> and <Rm> not both from R0-R7)
```

Decode for this encoding

```
let n : integer = UInt(N::Rn);
let m : integer = UInt(Rm);
let shift_t : SRTType = SRTType_LSL;
let shift_n : integer = 0;
if n < 8 && m < 8 then UnpredictableProcedure(); end;
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $n < 8$ && $m < 8$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The condition flags become UNKNOWN.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
1	1	1	0	1	0	1	1	1	0	1	1	Rn				(0)	imm3				1	1	1	imm2				stype		Rm																	
op1												S																		Rd																	

Encoding for the Rotate right with extend variant

Applies when (imm3 == 000 && imm2 == 00 && stype == 11)

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when (! (imm3 == 000 && imm2 == 00 && stype == 11))

```
CMP{<c>}{<q>} <Rn>, <Rm>, <shift> #<amount>
CMP{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1 or T2)
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> For the "A1 Rotate right with extend" and "A1 Shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For the "T1", "T3 Rotate right with extend", and "T3 Shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.
For the "T2" variant: is the first general-purpose source register, encoded in the "N:Rn" field.
- <Rm> For the "A1 Rotate right with extend" and "A1 Shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For the "T1", "T2", "T3 Rotate right with extend", and "T3 Shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the type of shift to be applied to the second source register, encoded in "stype":

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <amount> For the "A1 Shift or rotate by value" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
For the "T3 Shift or rotate by value" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    PSTATE.[N,Z,C,V] = nzcvc;
end;
```

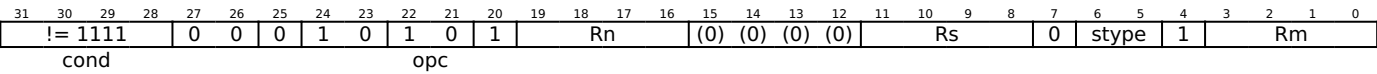
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

CMP (register-shifted register)

Compare (register-shifted register) subtracts a register-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

A1



Encoding

CMP{<c>} {<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SRTYPE = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <type> Is the type of shift to be applied to the second source register, encoded in "stype":

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR
- <Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let shift_n : integer = UInt(R(s)[7:0]);
  let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
  let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
  PSTATE.[N,Z,C,V] = nzcvc;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

CPS, CPSID, CPSIE

Change PE State changes one or more of the *PSTATE*.{A, I, F} interrupt mask bits and, optionally, the *PSTATE*.M mode field, without changing any other *PSTATE* bits.

CPS is treated as NOP if executed in User mode unless it is defined as being CONSTRAINED UNPREDICTABLE elsewhere in this section.

The PE checks whether the value being written to PSTATE.M is legal. See *Illegal changes to PSTATE.M*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	imod	M	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	A	I	F	0	mode				
														op														E			

Encoding for the Change mode variant

Applies when (imod == 00 && M == 1)

```
CPS{<q>} #<mode> // (Cannot be conditional)
```

Encoding for the Interrupt disable variant

Applies when (imod == 11 && M == 0)

```
CPSID{<q>} <iflags> // (Cannot be conditional)
```

Encoding for the Interrupt disable and change mode variant

Applies when (imod == 11 && M == 1)

```
CPSID{<q>} <iflags> , #<mode> // (Cannot be conditional)
```

Encoding for the Interrupt enable variant

Applies when (imod == 10 && M == 0)

```
CPSIE{<q>} <iflags> // (Cannot be conditional)
```

Encoding for the Interrupt enable and change mode variant

Applies when (imod == 10 && M == 1)

```
CPSIE{<q>} <iflags> , #<mode> // (Cannot be conditional)
```

Decode for all variants of this encoding

```
if mode != '00000' && M == '0' then UnpredictableProcedure(); end;
if (imod[1] == '1' && A::I::F == '000') || (imod[1] == '0' && A::I::F != '000') then
  UnpredictableProcedure();
end;
let enable : boolean = (imod == '10');
let disable : boolean = (imod == '11');
let changemode : boolean = (M == '1');
let pemode : bits(5) = mode;
let affectA : boolean = (A == '1');
let affectI : boolean = (I == '1');
let affectF : boolean = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If *imod* == '01', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If *imod* == '00' && *M* == '0', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If *mode* != '00000' && *M* == '0', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: changemode = TRUE.
- The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.

If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '0'`.
- The instruction behaves as if `A:I:F` has an UNKNOWN nonzero value.

If `imod<1> == '0' && A:I:F != '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '1'`.
- The instruction behaves as if `A:I:F == '000'`.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	A	I	F

op

Encoding for the Interrupt disable variant

Applies when `(im == 1)`

`CPSID{<q> <iflags> // (Not permitted in IT block)`

Encoding for the Interrupt enable variant

Applies when `(im == 0)`

`CPSIE{<q> <iflags> // (Not permitted in IT block)`

Decode for all variants of this encoding

```
if A::I::F == '000' then UnpredictableProcedure(); end;
let enable : boolean = (im == '0');
let disable : boolean = (im == '1');
let changemode : boolean = FALSE;
let affectA : boolean = (A == '1');
let affectI : boolean = (I == '1');
let affectF : boolean = (F == '1');
let pemode : bits(5) = ARBITRARY : bits(5);
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod	M	A	I	F	mode						

Encoding for the Change mode variant

Applies when `(imod == 00 && M == 1)`

`CPS{<q> #<mode> // (Not permitted in IT block)`

Encoding for the Interrupt disable variant

Applies when `(imod == 11 && M == 0)`

`CPSID.W <iflags> // (Not permitted in IT block)`

Encoding for the Interrupt disable and change mode variant

Applies when `(imod == 11 && M == 1)`

`CPSID{<q> <iflags>, #<mode> // (Not permitted in IT block)`

Encoding for the Interrupt enable variant

Applies when `(imod == 10 && M == 0)`

CPSIE.W <iflags> // (Not permitted in IT block)

Encoding for the Interrupt enable and change mode variant

Applies when (imod == 10 && M == 1)

CPSIE{<q>} <iflags>, #<mode> // (Not permitted in IT block)

Decode for all variants of this encoding

```
if imod == '00' && M == '0' then See("Hint instructions"); end;
if mode != '00000' && M == '0' then UnpredictableProcedure(); end;
if (imod[1] == '1' && A::I::F == '000') || (imod[1] == '0' && A::I::F != '000') then
    UnpredictableProcedure();
end;
let enable : boolean = (imod == '10');
let disable : boolean = (imod == '11');
let changemode : boolean = (M == '1');
let pemode : bits(5) = mode;
let affectA : boolean = (A == '1');
let affectI : boolean = (I == '1');
let affectF : boolean = (F == '1');
if imod == '01' || InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `changemode = TRUE`.
- The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.

If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '0'`.
- The instruction behaves as if A:I:F has an UNKNOWN nonzero value.

If `imod<1> == '0' && A:I:F != '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '1'`.
- The instruction behaves as if A:I:F == '000'.

Hint instructions: In encoding T2, if the imod field is 0b00 and the M bit is 0b0, a hint instruction is encoded. To determine which hint instruction, see [Branches and miscellaneous control](#).

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .
<mode>	Is the number of the mode to change to, in the range 0 to 31, encoded in the "mode" field.
<iflags>	Is a sequence of one or more of the following, specifying which interrupt mask bits are affected: a Sets the A bit in the instruction, causing the specified effect on PSTATE.A , the SError interrupt mask bit. i Sets the I bit in the instruction, causing the specified effect on PSTATE.I , the IRQ interrupt mask bit. f Sets the F bit in the instruction, causing the specified effect on PSTATE.F , the FIQ interrupt mask bit.

Operation

```
if CurrentInstrSet\(\) == InstrSet\_A32 then
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0'; end;
            if affectI then PSTATE.I = '0'; end;
            if affectF then PSTATE.F = '0'; end;
        end;
        if disable then
            if affectA then PSTATE.A = '1'; end;
            if affectI then PSTATE.I = '1'; end;
            if affectF then PSTATE.F = '1'; end;
        end;
        if changemode then
            // AArch32_WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32\_WriteModeByInstr(pemode);
        end;
    end;
else
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0'; end;
            if affectI then PSTATE.I = '0'; end;
            if affectF then PSTATE.F = '0'; end;
        end;
        if disable then
            if affectA then PSTATE.A = '1'; end;
            if affectI then PSTATE.I = '1'; end;
            if affectF then PSTATE.F = '1'; end;
        end;
        if changemode then
            // AArch32_WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32\_WriteModeByInstr(pemode);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CRC32

CRC32 performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_CRC32)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
!= 1111				0	0	0	1	0	sz		0	Rn				Rd				(0) (0)		0	(0)	0	1	0	0	Rm								
cond																					C															

Encoding for the CRC32B variant

Applies when (sz == 00)

CRC32B{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32H variant

Applies when (sz == 01)

CRC32H{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32W variant

Applies when (sz == 10)

CRC32W{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
if cond != '1110' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32;.

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1
(FEAT_CRC32)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	0	Rn				1	1	1	1	Rd				1	0	sz		Rm			
C																															

Encoding for the CRC32B variant

Applies when (sz == 00)

CRC32B{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32H variant

Applies when (sz == 01)

CRC32H{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32W variant

Applies when (sz == 10)

CRC32W{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `size = 32`;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields . A CRC32 instruction must be unconditional.
<Rd>	Is the general-purpose accumulator output register, encoded in the "Rd" field.
<Rn>	Is the general-purpose accumulator input register, encoded in the "Rn" field.
<Rm>	Is the general-purpose data source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    let acc : bits(32) = R(n);           // accumulator
    let val : bits(size) = R(m)[size-1:0]; // input value
    let poly : bits(32) = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)[31:0];
    let tempacc : bits(32+size) = BitReverse{32}(acc)::Zeros{size};
    let tempval : bits(32+size) = BitReverse{size}(val)::Zeros{32};
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R(d) = BitReverse{32}(Poly32Mod2{32+size}(tempacc XOR tempval, poly));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CRC32C

CRC32C performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_CRC32)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	0	1	0	sz		0	Rn				Rd				(0)	(0)	1	(0)	0	1	0	0	Rm				
cond																					C											

Encoding for the CRC32CB variant

Applies when (sz == 00)

CRC32CB{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32CH variant

Applies when (sz == 01)

CRC32CH{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32CW variant

Applies when (sz == 10)

CRC32CW{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
if cond != '1110' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32;.

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1
(FEAT_CRC32)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	1	Rn				1	1	1	1	Rd				1	0	sz		Rm			
C																															

Encoding for the CRC32CB variant

Applies when (sz == 00)

CRC32CB{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32CH variant

Applies when (sz == 01)

CRC32CH{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32CW variant

Applies when (sz == 10)

CRC32CW{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `size = 32`;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields . A CRC32C instruction must be unconditional.
<Rd>	Is the general-purpose accumulator output register, encoded in the "Rd" field.
<Rn>	Is the general-purpose accumulator input register, encoded in the "Rn" field.
<Rm>	Is the general-purpose data source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    let acc : bits(32) = R(n);           // accumulator
    let val : bits(size) = R(m)[size-1:0]; // input value
    let poly : bits(32) = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)[31:0];
    let tempacc : bits(32+size) = BitReverse{32}(acc)::Zeros{size};
    let tempval : bits(32+size) = BitReverse{size}(val)::Zeros{32};
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R(d) = BitReverse{32}(Poly32Mod2{32+size}(tempacc XOR tempval, poly));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution arising from data value prediction. For more information and details of the semantics, see *Consumption of Speculative Data Barrier (CSDB)*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	1	0	0
cond				R								imm4				imm12															

Encoding

CSDB{<c>}{<q>}

Decode for this encoding

```
if cond != '1110' then UnpredictableProcedure(); end; // CSDB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	1	0	0
																hint								option							

Encoding

CSDB{<c>}{<q>}

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c>

See *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    ConsumptionOfSpeculativeDataBarrier();
end;
```

DBG

DBG executes as a NOP. Arm deprecates any use of the DBG instruction.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	1	1	1	1	option			
cond				R				imm4																							

Encoding

DBG{<c>} {<q>} #<option>

Decode for this encoding

// DBG executes as a NOP. The 'option' field is ignored

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	1	1	1	1	option			
																hint															

Encoding

DBG{<c>} {<q>} #<option>

Decode for this encoding

// DBG executes as a NOP. The 'option' field is ignored

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c>

See [Standard assembler syntax fields](#).
- <q>

See [Standard assembler syntax fields](#).
- <option>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "option" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
end;
```

DCPS1

Debug Change PE State to EL1 allows the debugger to move the PE into EL1 from EL0 or to a specific mode at the current Exception level. DCPS1 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is implemented, EL2 is implemented and enabled in the current Security state, and any of:
 - EL2 is using AArch32 and HCR.TGE is set to 1.
 - EL2 is using AArch64 and HCR_EL2.TGE is set to 1.

When the PE executes DCPS1 at EL0, EL1 or EL3:

- If EL3 or EL1 is using AArch32, the PE enters SVC mode and LR_svc, SPSR_svc, DLR, and DSPSR become UNKNOWN. If DCPS1 is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL1 is using AArch64, the PE enters EL1 using AArch64, selects SP_EL1, and ELR_EL1, ESR_EL1, SPSR_EL1, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

When the PE executes DCPS1 at EL2 the PE does not change mode, and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN. For more information on the operation of the DCPS<n> instructions, see DCPS.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
imm4																imm10										opt					

Encoding

DCPS1

Decode for this encoding

// No additional decoding required.

Operation

```
if !Halted() then Undefined(); end;

if EL2Enabled() && PSTATE.EL == EL0 then
  let tge : bit = if ELUsingAArch32(EL2) then HCR().TGE else HCR_EL2().TGE;
  if tge == '1' then Undefined(); end;
end;

if PSTATE.EL != EL0 || ELUsingAArch32(EL1) then
  if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
  if PSTATE.EL != EL2 then
    AArch32_WriteMode(M32_Svc);
    PSTATE.E = SCTLRL().EE;
    if IsFeatureImplemented(FEAT_PAN) && SCTLRL().SPAN == '0' then PSTATE.PAN = '1'; end;
    Rmode(14, M32_Svc) = ARBITRARY : bits(32); // LR_svc
    SPSR_svc() = ARBITRARY : bits(32);
  else
    PSTATE.E = HSCTLR().EE;
    ELR_hyp() = ARBITRARY : bits(32);
    HSR() = ARBITRARY : bits(32);
    SPSR_hyp() = ARBITRARY : bits(32);
  end;

  DLR() = ARBITRARY : bits(32);
  DSPSR() = ARBITRARY : bits(32);
else // Targeting EL1 using AArch64
  AArch64_MaybeZeroRegisterUppers();
  MaybeZeroSVEUppers(EL1);
  PSTATE.nRW = '0';
  PSTATE.SP = '1';
  PSTATE.EL = EL1;
  if IsFeatureImplemented(FEAT_PAN) && SCTLRL_EL1().SPAN == '0' then PSTATE.PAN = '1'; end;
  if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;

  ELR_EL1() = ARBITRARY : bits(64);
  ESR_EL1() = ARBITRARY : bits(64);
  SPSR_EL1() = ARBITRARY : bits(64);

  DLR_EL0() = ARBITRARY : bits(64);
  DSPSR_EL0() = ARBITRARY : bits(64);

  // SCTLRL_EL1.IESB might be ignored in Debug state.
  if (IsFeatureImplemented(FEAT_IESB) && SCTLRL_EL1().IESB == '1' &&
    !ConstrainUnpredictableBool(Unpredictable_IESBinDebug)) then
    SynchronizeErrors();
  end;
end;

UpdateEDSCRFields(); // Update EDSCR PE state flags
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DCPS2

Debug Change PE State to EL2 allows the debugger to move the PE into EL2 from a lower Exception level.

DCPS2 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is not implemented.
- The PE is in Secure state and any of:
 - Secure EL2 is not implemented.
 - Secure EL2 is implemented and Secure EL2 is disabled.

When the PE executes DCPS2:

- If EL2 is using AArch32, the PE enters Hyp mode and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.
- If EL2 is using AArch64, the PE enters EL2 using AArch64, selects SP_EL2, and ELR_EL2, ESR_EL2, SPSR_EL2, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see [DCPS](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
imm4																imm10										opt					

Encoding

DCPS2

Decode for this encoding

```
if !HaveEL(EL2) then Undefined(); end;
```

Operation

```
if !Halted() || !EL2Enabled() then Undefined(); end;

if ELUsingAArch32(EL2) then
  AArch32_WriteMode(M32_Hyp);
  PSTATE.E = HSCTLR().EE;

  ELR_hyp() = ARBITRARY : bits(32);
  HSR() = ARBITRARY : bits(32);
  SPSR_hyp() = ARBITRARY : bits(32);

  DLR() = ARBITRARY : bits(32);
  DSPSR() = ARBITRARY : bits(32);
else
  // Targeting EL2 using AArch64
  AArch64_MaybeZeroRegisterUppers();
  MaybeZeroSVEUppers(EL2);
  PSTATE.nRW = '0';
  PSTATE.SP = '1';
  PSTATE.EL = EL2;
  if IsFeatureImplemented(FEAT_PAN) && SCTLR_EL2().SPAN == '0' && ELIsInHost(EL0) then
    PSTATE.PAN = '1';
  end;
  if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;

  ELR_EL2() = ARBITRARY : bits(64);
  ESR_EL2() = ARBITRARY : bits(64);
  SPSR_EL2() = ARBITRARY : bits(64);

  DLR_EL0() = ARBITRARY : bits(64);
  DSPSR_EL0() = ARBITRARY : bits(64);

  // SCTLR_EL2.IESB might be ignored in Debug state.
  if (IsFeatureImplemented(FEAT_IESB) && SCTLR_EL2().IESB == '1' &&
      !ConstrainUnpredictableBool(Unpredictable_IESBinDebug)) then
    SynchronizeErrors();
  end;
end;

UpdateEDSCRFields(); // Update EDSCR PE state flags
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DCPS3

Debug Change PE State to EL3 allows the debugger to move the PE into EL3 from a lower Exception level or to a specific mode at the current Exception level.
DCPS3 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL3 is not implemented.
- EDSCR.SDD is set to 1.

When the PE executes DCPS3:

- If EL3 is using AArch32, the PE enters Monitor mode and LR_mon, SPSR_mon, DLR and DSPSR become UNKNOWN. If DCPS3 is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL3 is using AArch64, the PE enters EL3 using AArch64, selects SP_EL3, and ELR_EL3, ESR_EL3, SPSR_EL3, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see [DCPS](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
imm4																imm10										opt					

Encoding

DCPS3

Decode for this encoding

```
if !HaveEL(EL3) then Undefined(); end;
```


Operation

```
if !Halted() || EDSCR().SDD == '1' then Undefined(); end;

if ELUsingAArch32(EL3) then
  let from_secure : boolean = (CurrentSecurityState() == SS_Secure);
  if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
  AArch32_WriteMode(M32_Monitor);
  if IsFeatureImplemented(FEAT_PAN) then
    if !from_secure then
      PSTATE.PAN = '0';
    elsif SCTLRL().SPAN == '0' then
      PSTATE.PAN = '1';
    end;
  end;
  PSTATE.E = SCTLRL().EE;

  LR_mon = ARBITRARY : bits(32);
  SPSR_mon() = ARBITRARY : bits(32);

  DLR() = ARBITRARY : bits(32);
  DSPSR() = ARBITRARY : bits(32);
else // Targeting EL3 using AArch64
  AArch64_MaybeZeroRegisterUppers();
  MaybeZeroSVEUppers(EL3);
  PSTATE.nRW = '0';
  PSTATE.SP = '1';
  PSTATE.EL = EL3;
  if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;

  ELR_EL3() = ARBITRARY : bits(64);
  ESR_EL3() = ARBITRARY : bits(64);
  SPSR_EL3() = ARBITRARY : bits(64);

  DLR_EL0() = ARBITRARY : bits(64);
  DSPSR_EL0() = ARBITRARY : bits(64);

  var sync_errors : boolean = IsFeatureImplemented(FEAT_IESB) && SCTLRL_EL3().IESB == '1';
  if IsFeatureImplemented(FEAT_DoubleFault) && EffectiveEA() == '1' && SCR_EL3().NMEA == '1' then
    sync_errors = TRUE;
  end;
  // SCTLRL_EL3.IESB might be ignored in Debug state.
  if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
  end;
  if sync_errors then SynchronizeErrors(); end;
end;

UpdateEDSCRFields(); // Update EDSCR PE state flags
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier (DMB)*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	1	option			

opcode

Encoding

DMB{<c>}{<q>}{<option>}

Decode for this encoding

// No additional decoding required

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	1	option			

opc

Encoding

DMB{<c>}{<q>}{<option>}

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c>
- For the "A1" variant: see *Standard assembler syntax fields*. Must be AL or omitted.
- For the "T1" variant: see *Standard assembler syntax fields*.
- <q>
- See *Standard assembler syntax fields*.

<option>

Specifies an optional limitation on the barrier operation. Values are:

SY	Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = 0b1111.
ST	Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. SYST is a synonym for ST. Encoded as option = 0b1110.
LD	Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1101.
ISH	Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b1011.
ISHST	Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b1010.
ISHL	Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1001.
NSH	Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = 0b0111.
NSHST	Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = 0b0110.
NSHL	Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0101.
OSH	Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b0011.
OSHST	Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b0010.
OSHL	Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0001.

For more information on whether an access is before or after a barrier instruction, see [Data Memory Barrier \(DMB\)](#). All other encodings of option are reserved. All unsupported and reserved options must execute as a full system DMB operation, but software must not rely on this behavior.

Note

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var types : MReqTypes;
    case option[1:0] of
        when '00' => types = MReqTypes_All;
        when '01' => types = MReqTypes_Reads;
        when '10' => types = MReqTypes_Writes;
        when '11' => types = MReqTypes_All;
    end;

    DataMemoryBarrier(types);
end;
```

DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see [Data Synchronization Barrier \(DSB\)](#).
An AArch32 DSB instruction does not require the completion of any AArch64 TLB maintenance instructions, regardless of the nXS qualifier, appearing in program order before the AArch32 DSB.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	!= 0x00			
																								opcode				option			

Encoding

DSB{<c>} {<q>} {<option>}

Decode for this encoding

// No additional decoding required

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	!= 0x00			
																								opc				option			

Encoding

DSB{<c>} {<q>} {<option>}

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c>For the "A1" variant: see [Standard assembler syntax fields](#). Must be AL or omitted.
- For the "T1" variant: see [Standard assembler syntax fields](#).
- <q>See [Standard assembler syntax fields](#).

<option>

Specifies an optional limitation on the barrier operation. Values are:

SY	Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = 0b1111.
ST	Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. SYST is a synonym for ST. Encoded as option = 0b1110.
LD	Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1101.
ISH	Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b1011.
ISHST	Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b1010.
ISHL	Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1001.
NSH	Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = 0b0111.
NSHST	Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = 0b0110.
NSHL	Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0101.
OSH	Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b0011.
OSHST	Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b0010.
OSHL	Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0001.

For more information on whether an access is before or after a barrier instruction, see [Data Synchronization Barrier \(DSB\)](#). All other encodings of option are reserved, other than the values 0b0000 and 0b0100. All unsupported and reserved options must execute as a full system DSB operation, but software must not rely on this behavior.

Note

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB. The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var nXS : boolean;
    if IsFeatureImplemented(FEAT_XS) then
        nXS = (PSTATE.EL IN {EL0, EL1} && !ELUsingAArch32(EL2) &&
            IsHCRXEL2Enabled() && HCRX_EL2().FnXS == '1');
    else
        nXS = FALSE;
    end;
    var scope : MBMaintenanceScope;
    var types : MBRegTypes;
    case option[1:0] of
        when '00' =>
            types = MBRegTypes_All;
            scope = MBMaintenanceScope_OuterShareable;
        when '01' =>
            types = MBRegTypes_Reads;
            scope = MBMaintenanceScope_None;
        when '10' =>
            types = MBRegTypes_Writes;
            scope = MBMaintenanceScope_None;
        when '11' =>
            types = MBRegTypes_All;
            case option[3:2] of
                when '00' => scope = MBMaintenanceScope_OuterShareable;
                when '01' => scope = MBMaintenanceScope_Nonshareable;
                when '10' => scope = MBMaintenanceScope_InnerShareable;
                when '11' => scope = MBMaintenanceScope_OuterShareable;
            end;
    end;
    DataSynchronizationBarrier(scope, types, nXS);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR, EORS (immediate)

Bitwise Exclusive-OR (immediate) performs a bitwise exclusive-OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	0	0	1	S	Rn				Rd				imm12											
cond								opc																							

Encoding for the EOR variant

Applies when (S == 0)

EOR{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the EORS variant

Applies when (S == 1)

EORS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	0	1	0	0	S	Rn				0	imm3				Rd				imm8							
op1																																

Encoding for the EOR variant

Applies when (S == 0)

EOR{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the EORS variant

Applies when (S == 1 && Rd != 1111)

EORS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TEQ (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd>	For the "A1 EOR" and "A1 EORS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. For the "T1 EOR" and "T1 EORS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For the "A1 EOR" and "A1 EORS" variants: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T1 EOR" and "T1 EORS" variants: is the general-purpose source register, encoded in the "Rn" field.
<const>	For the "A1 EOR" and "A1 EORS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values. For the "T1 EOR" and "T1 EORS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                             else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) XOR imm32;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXCEPTIONRETURN(result);
        else
            ALUWRITEPC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR, EORS (register)

Bitwise Exclusive-OR (register) performs a bitwise exclusive-OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	0	1	S	Rn				Rd				imm5				stype		0	Rm				
cond								opc																							

Encoding for the EOR, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

EOR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the EOR, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

EOR{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the EORS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

EORS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the EORS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

EORS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	1	Rm			Rdn		
op															

Encoding

EOR{<c>}{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())

EORS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	0	0	S			Rn		(0)		imm3			Rd			imm2		stype			Rm		
op1																															

op1

Encoding for the EOR, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

EOR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the EOR, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

EOR{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

EOR<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the EORS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

EORS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the EORS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

EORS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

EORS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TEQ (register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> For the "A1 EOR, rotate right with extend", "A1 EOR, shift or rotate by value", "A1 EORS, rotate right with extend", and "A1 EORS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the EORS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the "T2 EOR, rotate right with extend", "T2 EOR, shift or rotate by value", "T2 EORS, rotate right with extend", and "T2 EORS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For the "A1 EOR, rotate right with extend", "A1 EOR, shift or rotate by value", "A1 EORS, rotate right with extend", and "A1 EORS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For the "T2 EOR, rotate right with extend", "T2 EOR, shift or rotate by value", "T2 EORS, rotate right with extend", and "T2 EORS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For the "A1 EOR, rotate right with extend", "A1 EOR, shift or rotate by value", "A1 EORS, rotate right with extend", and "A1 EORS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T1", "T2 EOR, rotate right with extend", "T2 EOR, shift or rotate by value", "T2 EORS, rotate right with extend", and "T2 EORS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the "A1 EOR, shift or rotate by value" and "A1 EORS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For the "T2 EOR, shift or rotate by value" and "T2 EORS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

In T32 assembly:

- Outside an IT block, if EORS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EORS <Rd>, <Rn> had been written
- Inside an IT block, if EOR<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EOR<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR, EORS (register-shifted register)

Bitwise Exclusive-OR (register-shifted register) performs a bitwise exclusive-OR of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	0	1	S	Rn				Rd				Rs				0	stype		1	Rm			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

EORS{<c>}{<q>}{<Rd>, <Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

EOR{<c>}{<q>}{<Rd>, <Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<Rs>	Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C[32](R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit[32](result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

ERET

Exception Return.

The PE branches to the address held in the register holding the preferred return address, and restores *PSTATE* from SPSR_<current_mode>.

The register holding the preferred return address is:

- *ELR_hyp*, when executing in Hyp mode.
- LR, when executing in a mode other than Hyp mode, User mode, or System mode.

The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.

Exception Return is CONSTRAINED UNPREDICTABLE in User mode and System mode.

In Debug state, the T1 encoding of ERET executes the DRPS operation.

It has encodings from the following instruction sets: A32 (*A1*) and T32 (*T1*) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	1	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	0	1	1	0	(1)	(1)	(1)	(0)	
cond										op0																					

Encoding

ERET{<c>}{<q>}

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	1	1	1	1	0	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	0	0	0	0	0	0
Rn																imm8															

Encoding

ERET{<c>}{<q>}

Decode for this encoding

```
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Halted() then
        if PSTATE.M IN {M32_User, M32_System} then
            UnpredictableProcedure(); // UNDEFINED or NOP
        else
            let new_pc_value : bits(32) = if PSTATE.EL == EL2 then ELR_hyp() else R(14);
            AArch32_ExceptionReturn(new_pc_value, SPSR_curr());
        end;
    else // Perform DRPS operation in Debug state
        if PSTATE.M == M32_User then
            Undefined();
        elseif PSTATE.M == M32_System then
            UnpredictableProcedure(); // UNDEFINED or NOP
        else
            SynchronizeContext();
            DebugRestorePSR();
        end;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR and VDISR. This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. For more information, see *RAS PE architecture* and Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture (ARM IHI 0100).

If *FEAT_RAS* is not implemented, this instruction executes as a NOP.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1
(FEAT_RAS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	0	0	0
cond				R				imm4				imm12																			

Encoding

ESB{<c>}{<q>}

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then ExecuteAsNOP(); end;
if cond != '1110' then UnpredictableProcedure(); end; // ESB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1
(FEAT_RAS)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	0	0	0
																hint								option							

Encoding

ESB{<c>}{<q>}

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then ExecuteAsNOP(); end;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

Assembler Symbols

- <c>
- See *Standard assembler syntax fields*.
- <q>
- See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    SynchronizeErrors();
    AArch32\_ESBOperation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        AArch32\_vESBOperation();
    elseif IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 && !ELUsingAArch32(EL3) then
        AArch64\_dESBOperation();
    end;
    TakeUnmaskedSErrorInterrupts();
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

HLT

Halting breakpoint causes a software breakpoint to occur.
Halting breakpoint is always unconditional, even inside an IT block.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0	0	imm12												0	1	1	1	imm4				
cond									opc																						

Encoding

HLT{<q>} {#}<imm>

Decode for this encoding

```
if EDSCR().HDE == '0' || !HaltingAllowed() then Undefined(); end;
if cond != '1110' then UnpredictableProcedure(); end; // HLT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	1	0	imm6					

Encoding

HLT{<q>} {#}<imm>

Decode for this encoding

```
if EDSCR().HDE == '0' || !HaltingAllowed() then Undefined(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <q>
- See [Standard assembler syntax fields](#). An HLT instruction must be unconditional.
- <imm>
- For the "A1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.
For the "T1" variant: is a 6-bit unsigned immediate, in the range 0 to 63, encoded in the "imm6" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

Operation

```
EncodingSpecificOperations();
let is_async : boolean = FALSE;
let fault : FaultRecord = NoFault();
Halt(DebugHalt HaltInstruction, is_async, fault);
```

HVC

Hypervisor Call causes a Hypervisor Call exception. For more information, see *Hypervisor Call (HVC) exception*. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- When EL3 is implemented and using AArch64, and SCR_EL3.HCE is set to 0.
- In Non-secure EL1 modes when EL3 is implemented and using AArch32, and SCR.HCE is set to 0.
- When EL3 is not implemented and either HCR_EL2.HCD is set to 1 or HCR.HCD is set to 1.
- When EL2 is not implemented.
- In Secure state, if EL2 is not enabled in the current Security state.
- In User mode.

The HVC instruction is CONSTRAINED UNPREDICTABLE in Hyp mode when EL3 is implemented and using AArch32, and SCR.HCE is set to 0.

On executing an HVC instruction, the *HSR, Hyp Syndrome Register* reports the exception as a Hypervisor Call exception, using the EC value 0x12, and captures the value of the immediate argument, see *Use of the HSR*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	0	0	imm12												0	1	1	1	imm4			
cond										opc																					

Encoding

HVC{<q>} {#}<imm16>

Decode for this encoding

```
if cond != '1110' then UnpredictableProcedure(); end;
let imm16 : bits(16) = imm12::imm4;
```

CONSTRAINED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	0	imm4				1	0	0	0	imm12											
o1												o2																			

Encoding

HVC{<q>} {#}<imm16>

Decode for this encoding

```
let imm16 : bits(16) = imm4::imm12;
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <q>
- See *Standard assembler syntax fields*. An HVC instruction must be unconditional.
- <imm16>
- For the "A1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

For the "T1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

Operation

```
EncodingSpecificOperations();
if PSTATE.EL IN {EL0, EL3} || !EL2Enabled() then
    Undefined();
end;

var hvc_enable : bit;
if HaveEL(EL3) then
    if ELUsingAArch32(EL3) then
        if SCR().HCE == '0' && PSTATE.EL == EL2 then UnpredictableProcedure(); end;
        hvc_enable = SCR().HCE;
    else
        hvc_enable = SCR_EL3().HCE;
    end;
else
    hvc_enable = if ELUsingAArch32(EL2) then NOT(HCR().HCD) else NOT(HCR_EL2().HCD);
end;

if hvc_enable == '0' then
    Undefined();
else
    AArch32_CallHypervisor(immm16);
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	0	option			
																												opcode			

Encoding

ISB{<c>} {<q>} {<option>}

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	0	option			
																												opc			

Encoding

ISB{<c>} {<q>} {<option>}

Decode for this encoding

```
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c>
- For the "A1" variant: see *Standard assembler syntax fields*. Must be AL or omitted.
- For the "T1" variant: see *Standard assembler syntax fields*.
- <q>
- See *Standard assembler syntax fields*.
- <option>
- Specifies an optional limitation on the barrier operation. Values are:
- SY
- Full system barrier operation, encoded as option = 0b1111. Can be omitted.
- All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
end;
```

IT

If-Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block are the same as, or the inverse of, the condition the `IT` instruction specifies for the first instruction in the block.

The `IT` instruction itself does not affect the condition flags, but the execution of the instructions in the IT block can change the condition flags.

16-bit instructions in the IT block, other than `CMP`, `CMN` and `TST`, do not set the condition flags. An `IT` instruction with the `AL` condition can change the behavior without conditional execution.

The architecture permits exception return to an instruction in the IT block only if the restoration of the `CPSR` restores `PSTATE.IT` to a state consistent with the conditions specified by the `IT` instruction. Any other exception return to an instruction in an IT block is `UNPREDICTABLE`. Any branch to a target instruction in an IT block is not permitted, and if such a branch is made it is `UNPREDICTABLE` what condition is used when executing that target instruction and any subsequent instruction in the IT block.

Many uses of the IT instruction are deprecated for performance reasons, and an implementation might include ITD controls that can disable those uses of IT, making them `UNDEFINED`.

For more information see [Conditional execution](#) and [Conditional instructions](#). The first of these sections includes more information about the ITD controls.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	firstcond				!= 0000			
mask															

Encoding

IT{<x>{<y>{<z>}}}{<q> <cond>

Decode for this encoding

```
if mask == '0000' then See("Related encodings"); end;
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then
  UnpredictableProcedure();
end;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1)`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The '1111' condition is treated as being the same as the '1110' condition, meaning always, and the `ITSTATE` state machine is progressed in the same way as for any other `cond_base` value.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Miscellaneous 16-bit instructions](#).

Assembler Symbols

<x>	The condition for the second instruction in the IT block. If omitted, the "mask" field is set to 0b1000. If present it is encoded in the "mask[3]" field:
T	firstcond[0]
E	NOT firstcond[0]
<y>	The condition for the third instruction in the IT block. If omitted and <x> is present, the "mask[2:0]" field is set to 0b100. If <y> is present it is encoded in the "mask[2]" field:
T	firstcond[0]
E	NOT firstcond[0]
<z>	The condition for the fourth instruction in the IT block. If omitted and <y> is present, the "mask[1:0]" field is set to 0b10. If <z> is present, the "mask[0]" field is set to 1, and it is encoded in the "mask[1]" field:
T	firstcond[0]
E	NOT firstcond[0]

<q> See *Standard assembler syntax fields*.

<cond> The condition for the first instruction in the IT block, encoded in the "firstcond" field. See *Condition codes* for the range of conditions available, and the encodings.

The conditions specified in an `IT` instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check `IT` instruction syntax for validity but do not generate assembled instructions for them. See *Conditional instructions*.

Operation

```
EncodingSpecificOperations();  
AArch32_CheckITEnabled(mask);  
PSTATE.IT[7:0] = firstcond::mask;  
ShouldAdvanceIT = FALSE;
```

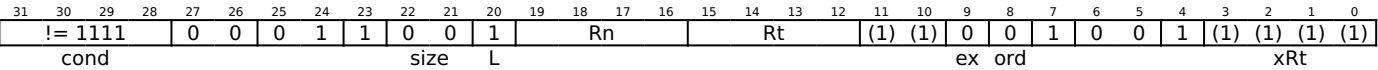
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDA

Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



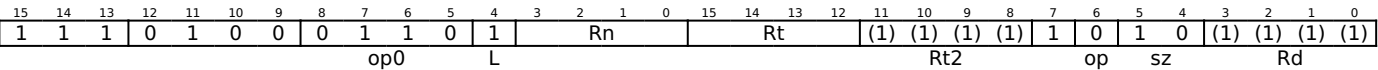
Encoding

```
LDA{<c>}{<q>}<Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
LDA{<c>}{<q>}<Rt>, [<Rn>]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    R(t) = MemO{32}(address);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDAB

Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	0	1	Rn				Rt				(1)	(1)	0	0	1	0	0	1	(1)	(1)	(1)	(1)
cond				size L								ex ord								xRt											

Encoding

LDAB{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				Rt				(1) (1) (1) (1)				1	0	0	0	(1) (1) (1) (1)			
op0												L	Rt2								op				sz	Rd					

Encoding

LDAB{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectoral Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    R(t) = ZeroExtend{32}(MemQ{8}(address));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

LDAEX

Load-Acquire Exclusive Word loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).
For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	0	1	Rn				Rt				(1) (1)		1	0	1	0	0	1	(1) (1) (1) (1)			
cond				size				L												ex				ord				xRt			

Encoding

LDAEX{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																	
1	1	1	0	1	0	0	0	1	1	0	1	Rn			Rt			(1)			(1)	(1)	(1)	1	1	1	0	(1)			(1)	(1)	(1)															
op0							L														Rt2							op							sz							Rd						

Encoding

LDAEX{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 4);
    R(t) = MemO{32}(address);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDAEXB

Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	0	1	Rn				Rt				(1)	(1)	1	0	1	0	0	1	(1)	(1)	(1)	(1)
cond				size				L												ex ord				xRt							

Encoding

LDAEXB{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				Rt				(1)	(1)	(1)	(1)	1	1	0	0	(1)	(1)	(1)	(1)
op0				L												Rt2				op				sz				Rd			

Encoding

LDAEXB{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 1);
    R(t) = ZeroExtend{32}{MemO{8}(address)};
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

LDAEXD

Load-Acquire Exclusive Doubleword loads a doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also acts as a barrier instruction with the ordering requirements described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0 0		0 1		1 1		0 1		1 1		Rn			Rt			(1) (1)		1 0		1 0		0 0		1 1		(1) (1)		(1) (1)			
cond								size				L												ex ord								xRt			

Encoding

LDAEXD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
if Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `t<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in [Using R15](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	0	0	0	1	1	0	1	Rn			Rt			Rt2			1	1	1	1	(1)	(1)	(1)	(1)							
op0												L				op												sz				Rd			

Encoding

LDAEXD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q>	See <i>Standard assembler syntax fields</i> .
<Rt>	For the "A1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14. For the "T1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For the "A1" variant: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>. For the "T1" variant: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 8);
    let value : bits(64) = MemO{64}(address);
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R(t) = if BigEndian(AccessType_GPR) then value[63:32] else value[31:0];
    R(t2) = if BigEndian(AccessType_GPR) then value[31:0] else value[63:32];
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

LDAEXH

Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
 - Causes the executing PE to indicate an active exclusive access in the local monitor.
- The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).
For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	1	1	Rn				Rt				(1)	(1)	1	0	1	0	0	1	(1)	(1)	(1)	(1)
cond				size				L												ex ord				xRt							

Encoding

LDAEXH{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				Rt				(1)	(1)	(1)	(1)	1	1	0	1	(1)	(1)	(1)	(1)
op0				L												Rt2				op				sz				Rd			

Encoding

LDAEXH{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 2);
    R(t) = ZeroExtend{32}(MemO{16}(address));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDAH

Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	1	Rn				Rt				(1)	(1)	0	0	1	0	0	1	(1)	(1)	(1)	(1)	
cond				size L								ex ord								xRt											

Encoding

LDAH{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	0	0	0	1	1	0	1	Rn				Rt				(1) (1) (1) (1)				1	0	0	1	(1) (1) (1) (1)							
op0												L				Rt2								op				sz				Rd			

Encoding

LDAH{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    R(t) = ZeroExtend{32}(MemQ{16}(address));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDC (immediate)

Load data to System register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to the *DBGDTRTXint* System register. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses, see *Memory accesses*.

In an implementation that includes EL2, the permitted LDC access to *DBGDTRTXint* can be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *HDCR.TDA*.

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	P	U	0	W	1	!= 1111				0	1	0	1	1	1	1	0	imm8							
cond				D				L				Rn				CRd				cp15											

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!

Encoding for the Unindexed variant

Applies when (P == 0 && U == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>], <option>

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDC (literal)"); end;
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	0	W	1	!= 1111				0	1	0	1	1	1	1	0	imm8							
				D				L				Rn				CRd				cp15											

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!

Encoding for the Unindexed variant

Applies when (P == 0 && U == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>], <option>

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDC (literal)"); end;
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
```

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see [LDC \(literal\)](#).
- +/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+
- <imm> Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.
- <option> Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
  let address : bits(32) = if index then offset_addr else R(n);

  // System register write to DBGDTRTXint.
  AArch32_SysRegWriteM(cp, ThisInstr(), address);

  if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDC (literal)

Load data to System register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to the *DBGDTRTXint* System register. For information about memory accesses, see *Memory accesses*.

In an implementation that includes EL2, the permitted LDC access to *DBGDTRTXint* can be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *HDCR.TDA*.

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	P	U	0	W	1	1	1	1	1	0	1	0	1	1	1	1	0	imm8							
cond				D				L				Rn				CRd				cp15											

Encoding

Applies when $(!(P == 0 \ \&\& \ U == 0 \ \&\& \ W == 0))$

LDC{<c>}{<q>} p14, c5, <label> // (Preferred syntax)

LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]

LDC{<c>}{<q>} p14, c5, [PC], <option>

Decode for this encoding

```
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $\bar{W} == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	0	W	1	1	1	1	1	0	1	0	1	1	1	1	0	imm8							
				D				L				Rn				CRd				cp15											

Encoding

Applies when $(!(P == 0 \ \&\& \ U == 0 \ \&\& \ W == 0))$

LDC{<c>}{<q>} p14, c5, <label> // (Preferred syntax)

LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]

Decode for this encoding

```
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $\bar{W} == '1' \ || \ P == '0'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes as LDC (immediate) with writeback to the PC. The instruction is handled as described in *Using R15*.

Assembler Symbols

<c>

See *Standard assembler syntax fields*.

<q>	See Standard assembler syntax fields .						
<label>	The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE (encoded as U == 1). If the offset is negative, imm32 is equal to minus the offset and add == FALSE (encoded as U == 0).						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:						
	<table> <tr> <th>U</th><th>+/-</th></tr> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.						
<option>	Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.						

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = (if add then (AlignDownSize(PC32(), 4) + imm32)
                                else (AlignDownSize(PC32(), 4) - imm32));
    let address : bits(32) = if index then offset_addr else AlignDownSize(PC32(), 4);

    // System register write to DBGDTRTXint.
    AArch32_SysRegWriteM(cp, ThisInstr(), address);
end;
```

Operational information

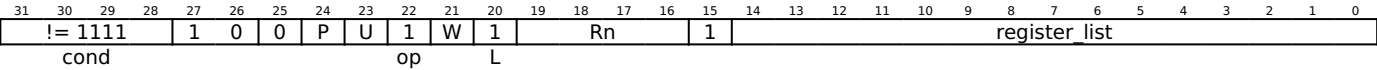
This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDM (exception return)

Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The *SPSR* of the current mode is copied to the *CPSR*. An address adjusted by the size of the data loaded can optionally be written back to the base register. The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address. The PE checks the encoding that is copied to the *CPSR* for an illegal return event. See *Illegal return events from AArch32 state*. Load Multiple (exception return) is:

- UNDEFINED in Hyp mode.
- UNPREDICTABLE in debug state, and in User mode and System mode.

A1



Encoding

```
LDM{<amode>}{<c>}{<q>} <Rn>{!}, <registers_with_pc>^
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(15) = register_list;
let wback : boolean = (W == '1');
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all the loads using the specified addressing mode and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during the execution of this instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<amode>	is one of:
DA	Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
FA	Full Ascending. For this instruction, a synonym for DA.
DB	Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
EA	Empty Ascending. For this instruction, a synonym for DB.
IA	Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
FD	Full Descending. For this instruction, a synonym for IA.
IB	Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.
ED	Empty Descending. For this instruction, a synonym for IB.
<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers_with_pc> Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must be specified in the register list, and the instruction causes a branch to the address (data) loaded into the PC. See also [Encoding of lists of general-purpose registers and the PC](#).

Instructions with similar syntax but without the PC included in the registers list are described in [LDM \(User registers\)](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        Undefined();
    elseif PSTATE.M IN {M32_User, M32_System} then
        UnpredictableProcedure(); // UNDEFINED or NOP
    else
        let length : integer = 4*BitCount(registers) + 4;
        var address : bits(32) = if increment then R(n) else R(n)-length;
        if wordhigher then address = address+4; end;
        var data : bits(32);

        for i = 0 to 14 do
            if registers[i] == '1' then
                if i != n then
                    R(i) = MemS{32}(address);
                else
                    data = MemS{32}(address);
                end;
                address = address + 4;
            end;
        end;
        let new_pc_value : bits(32) = MemS{32}(address);

        if wback && registers[n] == '1' then
            R(n) = ARBITRARY : bits(32);
        end;
        if wback && registers[n] == '0' then
            R(n) = if increment then R(n)+length else R(n)-length;
        end;
        if !wback && registers[n] == '1' then
            R(n) = data;
        end;

        AArch32_ExceptionReturn(new_pc_value, SPSR_curr());
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User, M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

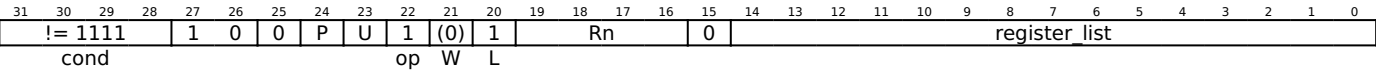
LDM (User registers)

In an EL1 mode other than System mode, Load Multiple (User registers) loads multiple User mode registers from consecutive memory locations using an address from a base register. The registers loaded cannot include the PC. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Load Multiple (User registers) is UNDEFINED in Hyp mode, and UNPREDICTABLE in User and System modes.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#).

A1



Encoding

LDM{<amode>}{<c>}{<q>} <Rn>, <registers_without_pc>^

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(15) = register_list;
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<amode>	is one of:
DA	Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
FA	Full Ascending. For this instruction, a synonym for DA.
DB	Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
EA	Empty Ascending. For this instruction, a synonym for DB.
IA	Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
FD	Full Descending. For this instruction, a synonym for IA.
IB	Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.
ED	Empty Descending. For this instruction, a synonym for IB.
<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<registers_without_pc>	Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the LDM instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list. See also Encoding of lists of general-purpose registers and the PC .

Instructions with similar syntax but with the PC included in <registers_without_pc> are described in [LDM \(exception return\)](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then Undefined();
    elsif PSTATE.M IN {M32_User, M32_System} then UnpredictableProcedure();
    else
        let length : integer = 4*BitCount(registers);
        var address : bits(32) = if increment then R(n) else R(n)-length;
        if wordhigher then address = address+4; end;
        for i = 0 to 14 do
            if registers[i] == '1' then // Load User mode register
                Rmode(i, M32_User) = MemS{32}(address); address = address + 4;
            end;
        end;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User, M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDM, LDMIA, LDMFD

Load Multiple (Increment After, Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#). The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#). Related system instructions are [LDM \(User registers\)](#) and [LDM \(exception return\)](#).

This instruction is used by the alias [POP \(multiple registers\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	0	0	0	1	0	W	1	Rn				register_list															
cond				P				U				op				L															

Encoding

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	Rn			register_list							
L															

Encoding

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = '00000000'::register_list;
let wback : boolean = (registers[n] == '0');
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

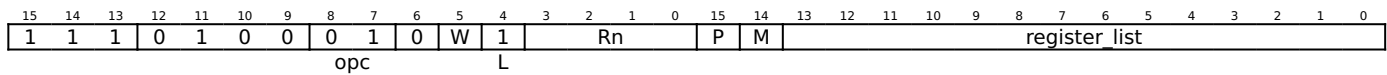
CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T2



Encoding

```
LDM{IA}{<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be represented in T1)
LDMFD{<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack, if <Rn>, '!' and <registers> can be represented in
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' && !InitBlock() && !LastInitBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

IA	Is an optional suffix for the Increment After form.
<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

!

For the "A1" and "T2" variants: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

For the "T1" variant: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if <Rn> is included in <registers>, otherwise it must be present.

<registers>

For the "A1" variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list.

For the "T1" variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field.

For the "T2" variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Alias Conditions

Alias	Of variant	Is preferred when
POP (multiple registers)	A1	<code>W == '1' && Rn == '1101' && BitCount(register_list) > 1</code>
POP (multiple registers)	T2	<code>W == '1' && Rn == '1101' && BitCount(P :: M :: register_list) > 1</code>
POP (multiple registers)	T2	<code>BitCount(P :: M :: register_list) > 1</code>

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n);
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address);
            else
                data = MemS{32}(address);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) + 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;
end;

```

Operational information

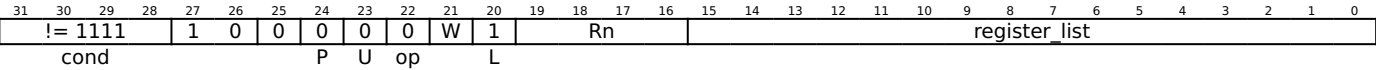
This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDMDA, LDMFA

Load Multiple Decrement After (Full Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).
Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#). The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#). Related system instructions are [LDM \(User registers\)](#) and [LDM \(exception return\)](#).

A1



Encoding

```
LDMDA{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
LDMFA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `BitCount(registers) < 1`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.
- If `wback && registers[n] == '1'`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.
- ! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
- <registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = (R(n) - 4*BitCount(registers)) + 4;
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address);
            else
                data = MemS{32}(address);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) - 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDMDB, LDMEA

Load Multiple Decrement Before (Empty Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#). The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#). Related system instructions are [LDM \(User registers\)](#) and [LDM \(exception return\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	0	0	1	0	0	W	1	Rn				register_list															
cond				P				U				L																			

Encoding

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
LDMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	W	1	Rn			P	M	register list														
opc							L																								

Encoding

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
LDMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' && !InitBlock() && !LastInitBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<code><c></code>	See Standard assembler syntax fields .
<code><q></code>	See Standard assembler syntax fields .
<code><Rn></code>	Is the general-purpose base register, encoded in the "Rn" field.
<code>!</code>	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<code><registers></code>	For the "A1" variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list. For the "T1" variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list: • The LR must not be in the list. • The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) - 4*BitCount(registers);
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address);
            else
                data = MemS{32}(address);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) - 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

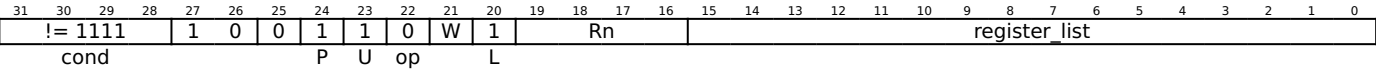
Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDMIB, LDMED

Load Multiple Increment Before (Empty Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).
Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#). The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#). Related system instructions are [LDM \(User registers\)](#) and [LDM \(exception return\)](#).

A1



Encoding

```
LDMIB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMED{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) + 4;
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address);
            else
                data = MemS{32}(address);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) + 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

This instruction is used by the alias [POP \(single register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) , [T3](#) and [T4](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	P	U	0	W	1	!= 1111				Rt				imm12											
cond				op2				op1				Rn																			

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, # {+/-} <imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn>], # {+/-} <imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDR{<c>}{<q>} <Rt>, [<Rn>, # {+/-} <imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
if P == '0' && W == '1' then See("LDRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if wback && n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	imm5					Rn			Rt		
B				L											

Encoding

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, # {+} <imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'00');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	Rt			imm8							

Encoding

```
LDR{<c>}{<q>}<Rt>, [SP{, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = 13;
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
1	1	1	1	1	1	0	0	0	1	1	0	1	!= 1111																												
size										L										Rn										imm12											

Encoding

```
LDR{<c>}{<q>}<Rt>, [<Rn> {, #{+}<imm>}]
```

```
LDR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)
```

Decode for this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm12};
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
if t == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	1	1	1	0	0	0	0	1	0	1	!= 1111				Rt				1				P	U	W	imm8												
size										L										Rn																			

Encoding for the Offset variant

Applies when (P == 1 && U == 0 && W == 0)

```
LDR{<c>}{<q>}<Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
LDR{<c>}{<q>}<Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDR{<c>}{<q>}<Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm8};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then
    UnpredictableProcedure();
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. For the "T1" and "T2" variants: is the general-purpose register to be transferred, encoded in the "Rt" field. For the "T3", "T4 Offset", "T4 Post-indexed", and "T4 Pre-indexed" variants: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.						
<Rn>	For the "A1 Offset", "A1 Post-indexed", "A1 Pre-indexed", "T3", "T4 Offset", "T4 Post-indexed", and "T4 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDR (literal). For the "T1" variant: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table> <tr> <th>U</th><th>+/-</th></tr> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field. For the "T1" variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4. For the "T2" variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4. For the "T3" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T4 Offset", "T4 Post-indexed", and "T4 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.						
+	Specifies the offset is added to the base register.						

Alias Conditions

Alias	Of variant	Is preferred when
POP (single register)	A1 Post-indexed	<code>P == '0' && U == '1' && W == '0' && Rn == '1101' && imm12 == '000000000100'</code>
POP (single register)	T4 Post-indexed	<code>Rn == '1101' && P == '0' && U == '1' && W == '1' && imm8 == '00000100'</code>

Operation

```
if CurrentInstrSet\(\) == InstrSet A32 then
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(32) = MemU{32}(address);
    if wback then R(n) = offset_addr; end;
    if t == 15 then
      if address[1:0] == '00' then
        LoadWritePC(data);
      else
        UnpredictableProcedure();
      end;
    else
      R(t) = data;
    end;
  end;
else
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(32) = MemU{32}(address);
    if wback then R(n) = offset_addr; end;
    if t == 15 then
      if address[1:0] == '00' then
        LoadWritePC(data);
      else
        UnpredictableProcedure();
      end;
    else
      R(t) = data;
    end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see [Memory accesses](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	1	0	P	U	0	W	1	1	1	1	1	Rt					imm12											
cond				op2				op1				Rn																				

Encoding

Applies when `(!(P == 0 && W == 1))`

```
LDR{<c>}{<q>} <Rt>, <label> // (Normal form)

LDR{<c>}{<q>} <Rt>, [PC, #<+/-><imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `wback = FALSE`;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in [LDR \(immediate\)](#). The instruction uses post-indexed addressing when `P == '0'` and uses pre-indexed addressing otherwise. The instruction is handled as described in [Using R15](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Rt			imm8							

Encoding

```
LDR{<c>}{<q>} <Rt>, <label> // (Normal form)
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let add : boolean = TRUE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	0	0	U	1	0	1	1	1	1	1	Rt					imm12											
size									L																							

Encoding

```
LDR{<c>}.W <Rt>, <label> // (Preferred syntax)

LDR{<c>}{<q>} <Rt>, <label> // (Preferred syntax, and <Rt>, <label> can be represented in T1)

LDR{<c>}{<q>} <Rt>, [PC, #<+/-><imm>] // (Alternative syntax)
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
if t == 15 && !InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	For the "A1" variant: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. For the "T1" variant: is the general-purpose register to be transferred, encoded in the "Rt" field. For the "T2" variant: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.						
<label>	For the "A1" and "T2" variants: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. For the "T1" variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field. For the "T2" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.						

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(), 4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    let data : bits(32) = MemU{32}(address);
    if t == 15 then
        if address[1:0] == '00' then
            LoadWritePC(data);
        else
            UnpredictableProcedure();
        end;
    else
        R(t) = data;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses, see [Memory accesses](#).
The T32 form of LDR (register) does not support register writeback.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	P	U	0	W	1	Rn				Rt				imm5				stype		0	Rm				
cond				op2				op1																							

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
if m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rm			Rn			Rt		
L				B		H									

Encoding

LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTYPE = SRTYPE.LSL;
let shift_n : integer = 0;
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	1	1	1	0	0	0	0	1	0	1	!= 1111				Rt				0				0	0	0	0	0	imm2		Rm								
size												L				Rn																							

Encoding

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
if t == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This branch is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. For the "T1" variant: is the general-purpose register to be transferred, encoded in the "Rt" field. For the "T2" variant: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.						
<Rn>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant. For the "T1" and "T2" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						
<shift>	The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register .						
+	Specifies the index register is added to the base register.						
<imm>	If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.						

Operation

```
if CurrentInstrSet() == InstrSet A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(32) = MemU{32}(address);
    if wback then R(n) = offset_addr; end;
    if t == 15 then
      if address[1:0] == '00' then
        LoadWritePC(data);
      else
        UnpredictableProcedure();
      end;
    else
      R(t) = data;
    end;
  end;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = (R(n) + offset);
    let address : bits(32) = offset_addr;
    let data : bits(32) = MemU{32}(address);
    if t == 15 then
      if address[1:0] == '00' then
        LoadWritePC(data);
      else
        UnpredictableProcedure();
      end;
    else
      R(t) = data;
    end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	P	U	1	W	1	!= 1111				Rt				imm12											
cond				op2				op1				Rn																			

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #<+/->imm}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRB{<c>}{<q>} <Rt>, [<Rn>], #<+/->imm

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRB{<c>}{<q>} <Rt>, [<Rn>, #<+/->imm]!

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRB (literal)"); end;
if P == '0' && W == '1' then See("LDRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `wback && n == t`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	imm5					Rn			Rt		
B				L											

Encoding

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #<+>imm}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	0	0	1	0	0	1	!= 1111				!= 1111				imm12												
size												L	Rn				Rt															

Encoding

```
LDRB{<c>}{<q>}<Rt>, [<Rn> {, #<+><imm>}]

LDRB{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```
if Rt == '1111' then See("PLD"); end;
if Rn == '1111' then See("LDRB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	0	0	0	0	0	1	!= 1111					Rt				1	P	U	W	imm8							
size												L				Rn																

Encoding for the Offset variant

Applies when (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRB{<c>}{<q>}<Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
LDRB{<c>}{<q>}<Rt>, [<Rn>], #<+/-><imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRB{<c>}{<q>}<Rt>, [<Rn>, #<+/-><imm>]!
```

Decode for all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("PLD, PLDW (immediate)"); end;
if Rn == '1111' then See("LDRB (literal)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRBT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	For the "A1 Offset", "A1 Post-indexed", "A1 Pre-indexed", "T2", "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRB (literal) . For the "T1" variant: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
For the "T1" variant: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.
For the "T2" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
For the "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

+ Specifies the offset is added to the base register.

Operation

```
if CurrentInstrSet() == InstrSet A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = ZeroExtend{32}(MemU{8}(address));
    if wback then R(n) = offset_addr; end;
  end;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = ZeroExtend{32}(MemU{8}(address));
    if wback then R(n) = offset_addr; end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
!= 1111				0	1	0	P	U	1	W	1	1	1	1	1	Rt					imm12															
cond				op2				op1				Rn																								

Encoding

Applies when (! (P == 0 && W == 1))

```
LDRB{<c>}{<q>}<Rt>, <label> // (Normal form)

LDRB{<c>}{<q>}<Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRBT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{imm12};
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If *wback*, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in *LDRB (immediate)*. The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15*.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
1	1	1	1	1	1	0	0	0	U	0	0	1	1	1	1	1	!= 1111					imm12																									
size																L																Rt															

Encoding

```
LDRB{<c>}{<q>}<Rt>, <label> // (Preferred syntax)

LDRB{<c>}{<q>}<Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
if Rt == '1111' then See("PLD"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{imm12};
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<label>	The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
For the "T1" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    R(t) = ZeroExtend{32}(MemU{8}(address));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	P	U	1	W	1	Rn				Rt				imm5				stype		0	Rm				
cond								op2				op1																			

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	Rm			Rn			Rt		
L				B		H									

Encoding

LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	0	0	0	0	0	1	!= 1111				!= 1111				0	0	0	0	0	0	imm2		Rm							
size												L				Rn				Rt															

Encoding

```
LDRB{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
LDRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rt == '1111' then See("PLD"); end;
if Rn == '1111' then See("LDRB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SType = SType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant. For the "T1" and "T2" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						
<shift>	The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register .						
+	Specifies the index register is added to the base register.						
<imm>	If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.						

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let offset : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
  let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
  let address : bits(32) = if index then offset_addr else R(n);
  R(t) = ZeroExtend{32}(MemU{8}(address));
  if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDRBT

Load Register Byte Unprivileged loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	0	U	1	1	1	Rn				Rt				imm12											
cond				P				op2				W				op1															

Encoding

LDRBT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	U	1	1	1	Rn				Rt				imm5				styp		0	Rm				
cond				P				op2				W				op1															

Encoding

LDRBT{<c>}{<q>} <Rt>, [<Rn>], {<+/-><Rm>{, <shift>}}

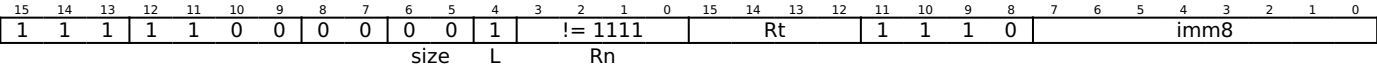
Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `n == t && n != 15`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

```
LDRBT{<c>}{<q>}<Rt>, [<Rn> {, #(<+>)<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("LDRB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> For the "A1" variant: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
For the "A2" and "T1" variants: is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.
- +/- For the "A1" variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":
- | U | +/- |
|---|-----|
| 0 | - |
| 1 | + |
- For the "A2" variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":
- | U | +/- |
|---|-----|
| 0 | - |
| 1 | + |
- <imm> For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
For the "T1" variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.
- <Rm> Is the general-purpose index register, encoded in the "Rm" field.
- <shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).
- +
- Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;           // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                             else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    R(t) = ZeroExtend{32}(MemU_unpriv{8}(address));
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRB (immediate).

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	0	!= 1111				Rt				imm4H				1	1	0	1	imm4L			
cond												o1				Rn												op2			

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #<+/-><imm>]!

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRD (literal)"); end;
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if wback && (n == t || n == t2) then UnpredictableProcedure(); end;
if t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `t<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when `Rt == '1111'`.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	P	U	1	W	1	!= 1111				Rt				Rt2				imm8							
L												Rn																			

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
LDRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '0' then See("Related encodings"); end;
if Rn == '1111' then See("LDRD (literal)"); end;
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if wback && (n == t || n == t2) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If wback && (n == t || n == t2), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: *Load/store dual, load/store exclusive, table branch*.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
<Rt>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14. For the "T1 Offset", "T1 Post-indexed", and "T1 Pre-indexed" variants: is the first general-purpose register to be transferred, encoded in the "Rt" field.						
<Rt2>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the second general-purpose register to be transferred. This register must be <R(t+1)>. For the "T1 Offset", "T1 Post-indexed", and "T1 Pre-indexed" variants: is the second general-purpose register to be transferred, encoded in the "Rt2" field.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. For PC use see <i>LDRD (literal)</i> .						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1 Offset", "T1 Post-indexed", and "T1 Pre-indexed" variants: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then R(n) + imm32 else R(n) - imm32;
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        let data : bits(64) = MemA{64}(address);
        if BigEndian(AccessType GPR) then
            R(t) = data[63:32];
            R(t2) = data[31:0];
        else
            R(t) = data[31:0];
            R(t2) = data[63:32];
        end;
    else
        let data1 : bits(32) = MemA{32}(address);
        let data2 : bits(32) = MemA{32}(address+4);
        R(t) = data1;
        R(t2) = data2;
    end;
    if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	(1)	U	1	(0)	0	1	1	1	1	Rt				imm4H				1	1	0	1	imm4L			
cond				P				W				o1				Rn								op2							

Encoding

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // (Normal form)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let add : boolean = (U == '1');
if t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `t<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when `Rt == '1111'`.

If `P == '0' || W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as if `P == 1` and `W == 0`.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	P	U	1	W	1	1	1	1	1	Rt				Rt2				imm8							
L																															

Encoding

Applies when `(!(P == 0 && W == 0))`

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // (Normal form)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '0' then See("Related encodings"); end;
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
if W == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

If `W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses post-indexed addressing when `P == '0'` and uses pre-indexed addressing otherwise. The instruction is handled as described in [Using R15](#).

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Load/Store dual](#), [Load/Store-Exclusive](#), [Load-Acquire/Store-Release](#), [table branch](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	For the "A1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14. For the "T1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field.						
<Rt2>	For the "A1" variant: is the second general-purpose register to be transferred. This register must be <Rt+1>. For the "T1" variant: is the second general-purpose register to be transferred, encoded in the "Rt2" field.						
<label>	For the "A1" variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. For the "T1" variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1" variant: is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.						

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = (if add then (AlignDownSize(PC32(), 4) + imm32)
                                else (AlignDownSize(PC32(), 4) - imm32));
    if IsAlignedSize(address, 8) then
        let data : bits(64) = MemA{64}(address);
        if BigEndian(AccessType_GPR) then
            R(t) = data[63:32];
            R(t2) = data[31:0];
        else
            R(t) = data[31:0];
            R(t2) = data[63:32];
        end;
    else
        let data1 : bits(32) = MemA{32}(address);
        let data2 : bits(32) = MemA{32}(address+4);
        R(t) = data1;
        R(t2) = data2;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDRD (register)

Load Register Dual (register) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	0	W	0	Rn				Rt				(0)	(0)	(0)	(0)	1	1	0	1	Rm			
cond				o1								op2																			

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>], {+/-}<Rm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!

Decode for all variants of this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if t2 == 15 || m == 15 || m == t || m == t2 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t || n == t2) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If `m == t || m == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads register Rm with an UNKNOWN value.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `t<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when `Rt == '1111'`.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.						
<Rt2>	Is the second general-purpose register to be transferred. This register must be <R(t+1)>.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.						
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + R(m)) else (R(n) - R(m));
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        let data : bits(64) = MemA{64}(address);
        if BigEndian(AccessType GPR) then
            R(t) = data[63:32];
            R(t2) = data[31:0];
        else
            R(t) = data[31:0];
            R(t2) = data[63:32];
        end;
    else
        let data1 : bits(32) = MemA{32}(address);
        let data2 : bits(32) = MemA{32}(address+4);
        R(t) = data1;
        R(t2) = data2;
    end;

    if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDREX

Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	0	1	Rn				Rt				(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond				size				L												ex				ord				xRt			

Encoding

LDREX{<c>}{<q>} <Rt>, [<Rn> {, {#}<imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = Zeros{}; // Zero offset
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	1	Rn				Rt				(1)	(1)	(1)	(1)	imm8							
				op0				L												Rd											

Encoding

LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<imm>	For the "A1" variant: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted. For the "T1" variant: the immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n) + imm32;
    AArch32_SetExclusiveMonitors(address,4);
    R(t) = MemA{32}(address);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDREXB

Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	0	1	Rn				Rt				(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond				size				L												ex				ord				xRt			

Encoding

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				Rt				(1)	(1)	(1)	(1)	0	1	0	0	(1)	(1)	(1)	(1)
				op0				L								Rt2								sz				Rd			

Encoding

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c>
- See [Standard assembler syntax fields](#).
- <q>
- See [Standard assembler syntax fields](#).
- <Rt>
- Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn>
- Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address,1);
    R(t) = ZeroExtend{32}(MemA{8}(address));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDREXD

Load Register Exclusive Doubleword derives an address from a base register value, loads a 64-bit doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	1	Rn				Rt				(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond				size				L												ex ord				xRt							

Encoding

LDREXD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
if Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `t<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in [Using R15](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				Rt				Rt2				0	1	1	1	(1)	(1)	(1)	(1)
				op0				L																sz				Rd			

Encoding

LDREXD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>	See Standard assembler syntax fields .
<Rt>	For the "A1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14. For the "T1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For the "A1" variant: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>. For the "T1" variant: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32\_SetExclusiveMonitors(address,8);
    let value : bits(64) = MemA{64}(address);

    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R(t) = if BigEndian(AccessType GPR) then value[63:32] else value[31:0];
    R(t2) = if BigEndian(AccessType GPR) then value[31:0] else value[63:32];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDREXH

Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	1	1	Rn				Rt				(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond				size L								ex ord								xRt											

Encoding

LDREXH{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				Rt				(1) (1) (1) (1)				0	1	0	1	(1) (1) (1) (1)			
op0												L				Rt2								sz				Rd			

Encoding

LDREXH{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address,2);
    R(t) = ZeroExtend{32}(MemA{16}(address));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	1	!= 1111				Rt				imm4H				1	0	1	1	imm4L			
cond											o1		Rn								op2										

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRH{<c>}{<q>}<Rt>, [<Rn> {, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRH{<c>}{<q>}<Rt>, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRH{<c>}{<q>}<Rt>, [<Rn>, #<+/-><imm>]!

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRH (literal)"); end;
if P == '0' && W == '1' then See("LDRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	imm5					Rn			Rt		
L															

Encoding

LDRH{<c>}{<q>}<Rt>, [<Rn> {, #<+><imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'0');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	1	!= 1111				!= 1111				imm12											
size											L	Rn				Rt															

Encoding

```
LDRH{<c>}{<q>}<Rt>, [<Rn> {, #<+><imm>}]
```

```
LDRH{<c>}.W<Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```
if Rt == '1111' then See("PLD (immediate)"); end;
if Rn == '1111' then See("LDRH (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	0	0	0	0	0	1	1	!= 1111				Rt				1	P	U	W	imm8									
size L												Rn																					

Encoding for the Offset variant

Applies when (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRH{<c>}{<q>}<Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
LDRH{<c>}{<q>}<Rt>, [<Rn>], #<+/-><imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRH{<c>}{<q>}<Rt>, [<Rn>, #<+/-><imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRH (literal)"); end;
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("PLDW (immediate)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRHT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	For the "A1 Offset", "A1 Post-indexed", "A1 Pre-indexed", "T2", "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRH (literal) . For the "T1" variant: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For the "T1" variant: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

For the "T2" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

+ Specifies the offset is added to the base register.

Operation

```
if CurrentInstrSet() == InstrSet A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = ZeroExtend{32}(data);
  end;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = ZeroExtend{32}(data);
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	1	1	1	1	Rt					imm4H				1	0	1	1	imm4L			
cond				o1								Rn												op2							

Encoding

Applies when (! (P == 0 && W == 1))

```
LDRH{<c>}{<q>}<Rt>, <label> // (Normal form)

LDRH{<c>}{<q>}<Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRHT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{imm4H::imm4L};
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If *wback*, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in *LDRH (immediate)*. The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15*.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	U	0	1	1	1	1	1	!= 1111				imm12											
size								L								Rt															

Encoding

```
LDRH{<c>}{<q>}<Rt>, <label> // (Preferred syntax)

LDRH{<c>}{<q>}<Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
if Rt == '1111' then See("PLD (literal)"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{imm12};
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <label> For the "A1" variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.
For the "T1" variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For the "T1" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    let data : bits(16) = MemU{16}(address);
    R(t) = ZeroExtend{32}(data);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	0	W	1	Rn				Rt				(0) (0) (0) (0) 1				0	1	1	Rm				
cond											o1					op2															

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	Rm			Rn			Rt		
				L	B	H									

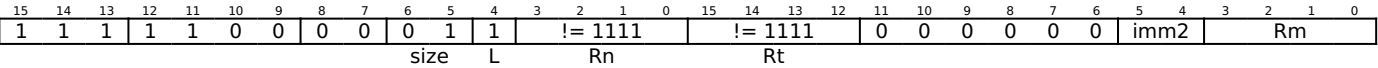
Encoding

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding

```
LDRH{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then See("LDRH (literal)"); end;
if Rt == '1111' then See("PLDW (register)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.
For the "T1" and "T2" variants: is the general-purpose base register, encoded in the "Rn" field.
- +/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+
- <Rm> Is the general-purpose index register, encoded in the "Rm" field.
- + Specifies the index register is added to the base register.
- <imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let offset : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
  let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
  let address : bits(32) = if index then offset_addr else R(n);
  let data : bits(16) = MemU{16}(address);
  if wback then R(n) = offset_addr; end;
  R(t) = ZeroExtend{32}(data);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDRHT

Load Register Halfword Unprivileged loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	U	1	1	1	Rn				Rt				imm4H				1	0	1	1	imm4L			
cond				P				W				o1												op2							

Encoding

LDRHT{<c>}{<q>}<Rt>, [<Rn>] {, #(<+/->)<imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : Bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	U	0	1	1	Rn				Rt				(0)	(0)	(0)	(0)	1	0	1	1	Rm			
cond				P				W				o1												op2							

Encoding

LDRHT{<c>}{<q>}<Rt>, [<Rn>], {<+/->}<Rm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : Bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

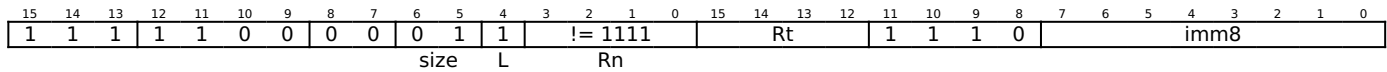
CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRHT{<c>}{<q>}<Rt>, [<Rn> {, #(+)<imm>}]

Decode for this encoding

```
if Rn == '1111' then See("LDRH (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.						
+/-	For the "A1" variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1"> <tr> <th>U</th><th>+/-</th></tr> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
	For the "A2" variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1"> <tr> <th>U</th><th>+/-</th></tr> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1" variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						
+	Specifies the offset is added to the base register.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end; // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    let data : bits(16) = MemU_unpriv{16}(address);
    if postindex then R(n) = offset_addr; end;
    R(t) = ZeroExtend{32}(data);
end;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The instruction executes as `LDRH` (immediate).

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	1	!= 1111				Rt				imm4H				1	1	0	1	imm4L			
cond											o1		Rn				op2														

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRSB{<c>}{<q>} <Rt>, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRSB{<c>}{<q>} <Rt>, [<Rn>, #<+/-><imm>]!

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRSB (literal)"); end;
if P == '0' && W == '1' then See("LDRSBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `wback && n == t`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	0	1	!= 1111				!= 1111				imm12											
size											Rn				Rt																

Encoding

LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

Decode for this encoding

```
if Rt == '1111' then See("PLI"); end;
if Rn == '1111' then See("LDRSB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	!= 1111				Rt				1	P	U	W	imm8							
size											Rn																				

Encoding for the Offset variant

Applies when (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRSB{<c>}{<q>}<Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
LDRSB{<c>}{<q>}<Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRSB{<c>}{<q>}<Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("PLI"); end;
if Rn == '1111' then See("LDRSB (literal)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRSBT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRSB (literal) .						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T2 Offset", "T2 Post-indexed", and "T2 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.						
+	Specifies the offset is added to the base register.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = SignExtend{32}{MemU{8}(address)};
    if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	0	P	U	1	W	1	1	1	1	1	Rt					imm4H				1	1	0	1	imm4L			
cond				o1					Rn														op2									

Encoding

Applies when (! (P == 0 && W == 1))

```
LDRSB{<c>}{<q>} <Rt>, <label> // (Normal form)

LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRSBT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{imm4H::imm4L};
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `wback = FALSE`;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in [LDRSB \(immediate\)](#). The instruction uses post-indexed addressing when `P == '0'` and uses pre-indexed addressing otherwise. The instruction is handled as described in [Using R15](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	!= 1111				imm12											
size												Rt																			

Encoding

```
LDRSB{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
if Rt == '1111' then See("PLI"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{imm12};
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <label> For the "A1" variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.
For the "T1" variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For the "T1" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    R(t) = SignExtend{32}(MemU{8}(address));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	0	W	1	Rn				Rt				(0)	(0)	(0)	(0)	1	1	0	1	Rm			
cond											o1				op2																

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRSB{<c>}{<q>}<Rt>, [<Rn>, {+/-}<Rm>]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRSB{<c>}{<q>}<Rt>, [<Rn>], {+/-}<Rm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRSB{<c>}{<q>}<Rt>, [<Rn>, {+/-}<Rm>]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRSBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRTYPE = SRTYPE\_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	Rm			Rn			Rt		
				L	B		H								

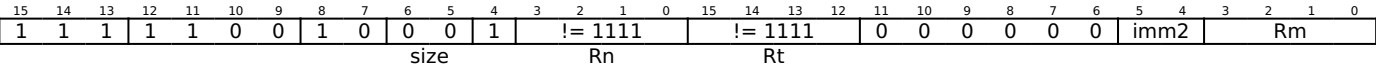
Encoding

LDRSB{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE\_LSL;
let shift_n : integer = 0;
```


T2



Encoding

```
LDRSB{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rt == '1111' then See("PLI"); end;
if Rn == '1111' then See("LDRSB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SType = SType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.
For the "T1" and "T2" variants: is the general-purpose base register, encoded in the "Rn" field.
- +/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+
- <Rm> Is the general-purpose index register, encoded in the "Rm" field.
- + Specifies the index register is added to the base register.
- <imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let offset : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
  let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
  let address : bits(32) = if index then offset_addr else R(n);
  R(t) = SignExtend{32}(MemU{8}(address));
  if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDRSBT

Load Register Signed Byte Unprivileged loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRSBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	U	1	1	1	Rn				Rt				imm4H				1	1	0	1	imm4L			
cond				P				W				o1												op2							

Encoding

LDRSBT{<c>}{<q>} <Rt>, [<Rn>] {, # {+/-} <imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : Bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if `bit[24] == '1'` and `bit[21] == '0'`. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	U	0	1	1	Rn				Rt				(0)	(0)	(0)	(0)	1	1	0	1	Rm			
cond				P				W				o1												op2							

Encoding

LDRSBT{<c>}{<q>} <Rt>, [<Rn>], {+/-} <Rm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : Bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	!= 1111				Rt				1	1	1	0	imm8							
size												Rn																			

Encoding

LDRSBT{<c>}{<q>} <Rt>, [<Rn> {, #(<+>)<imm>}]

Decode for this encoding

```
if Rn == '1111' then See("LDRSB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.						
+/-	For the "A1" variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1"> <thead> <tr> <th>U</th><th>+/-</th></tr> </thead> <tbody> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </tbody> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
	For the "A2" variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1"> <thead> <tr> <th>U</th><th>+/-</th></tr> </thead> <tbody> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </tbody> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1" variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						
+	Specifies the offset is added to the base register.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end; // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    R(t) = SignExtend{32}{MemU_unpriv{8}(address)};
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The instruction executes as `LDRSB` (immediate).

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	1	!= 1111				Rt				imm4H				1	1	1	imm4L				
cond											o1		Rn								op2										

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRSH{<c>}{<q>} <Rt>, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRSH{<c>}{<q>} <Rt>, [<Rn>, #<+/-><imm>]!

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
if P == '0' && W == '1' then See("LDRSHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `wback && n == t`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	1	1	!= 1111				!= 1111				imm12											
size											Rn					Rt															

Encoding

LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

Decode for this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
if Rt == '1111' then See("Related instructions"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	!= 1111				Rt				1	P	U	W	imm8							
size											Rn																				

Encoding for the Offset variant

Applies when (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRSH{<c>}{<q>}<Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
LDRSH{<c>}{<q>}<Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRSH{<c>}{<q>}<Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("Related instructions"); end;
if P == '1' && U == '1' && W == '0' then See("LDRSHT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).
Related instructions: [Load/store single](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRSH (literal) .						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T2 Offset", "T2 Post-indexed", and "T2 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.						
+	Specifies the offset is added to the base register.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = SignExtend{32}(data);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	0	P	U	1	W	1	1	1	1	1	Rt					imm4H				1	1	1	1	imm4L			
cond				o1					Rn					op2																		

Encoding

Applies when (! (P == 0 && W == 1))

```
LDRSH{<c>}{<q>} <Rt>, <label> // (Normal form)

LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
if P == '0' && W == '1' then See("LDRSHT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L)};
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `wback = FALSE`;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in [LDRSH \(immediate\)](#). The instruction uses post-indexed addressing when `P == '0'` and uses pre-indexed addressing otherwise. The instruction is handled as described in [Using R15](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	1	U	0	1	1	1	1	1	!= 1111				imm12											
size																Rt															

Encoding

```
LDRSH{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
if Rt == '1111' then See("Related instructions"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{(imm12)};
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).
Related instructions: [Load, signed \(literal\)](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <label> For the "A1" variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For the "T1" variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm> For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For the "T1" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(), 4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    let data : bits(16) = MemU{16}(address);
    R(t) = SignExtend{32}(data);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see [Memory accesses](#).
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111				0	0	0	P	U	0	W	1	Rn				Rt				(0)	(0)	(0)	(0)	1	1	1	1	Rm					
cond											o1				op2																		

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRSH{<c>}{<q>}<Rt>, [<Rn>, {+/-}<Rm>]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRSH{<c>}{<q>}<Rt>, [<Rn>], {+/-}<Rm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRSH{<c>}{<q>}<Rt>, [<Rn>, {+/-}<Rm>]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("LDRSHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	Rm			Rn			Rt		
				L	B		H								

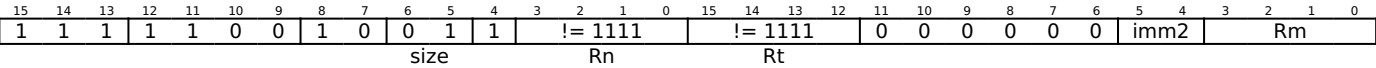
Encoding

LDRSH{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
if Rt == '1111' then See("Related instructions"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).
Related instructions: [Load/store, signed \(register offset\)](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant. For the "T1" and "T2" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						
+	Specifies the index register is added to the base register.						
<imm>	If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = SignExtend{32}(data);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

LDRSHT

Load Register Signed Halfword Unprivileged loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRSHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	U	1	1	1	Rn				Rt				imm4H				1	1	1	1	imm4L			
cond				P				W				o1												op2							

Encoding

LDRSHT{<c>}{<q>} <Rt>, [<Rn>] {, # {+/-} <imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : Bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	U	0	1	1	Rn				Rt				(0)	(0)	(0)	(0)	1	1	1	1	Rm			
cond				P				W				o1												op2							

Encoding

LDRSHT{<c>}{<q>} <Rt>, [<Rn>], {+/-} <Rm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : Bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	!= 1111				Rt				1	1	1	0	imm8							
size												Rn																			

Encoding

LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #(+)<imm>}]

Decode for this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.						
+/-	For the "A1" variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1"> <thead> <tr> <th>U</th><th>+/-</th></tr> </thead> <tbody> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </tbody> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
	For the "A2" variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1"> <thead> <tr> <th>U</th><th>+/-</th></tr> </thead> <tbody> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </tbody> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1" variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						
+	Specifies the offset is added to the base register.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end; // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    let data : bits(16) = MemU_unpriv{16}(address);
    if postindex then R(n) = offset_addr; end;
    R(t) = SignExtend{32}(data);
end;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The instruction executes as `LDRSH` (immediate).

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRT

Load Register Unprivileged loads a word from memory, and writes it to a register. For information about memory accesses see [Memory accesses](#). The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	0	U	0	1	1	Rn				Rt				imm12											
cond				P				op2				W				op1															

Encoding

LDRT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	U	0	1	1	Rn				Rt				imm5					stype	0	Rm				
cond				P				op2				W	op1																		

Encoding

LDRT{<c>}{<q>} <Rt>, [<Rn>], {<+/-><Rm>{, <shift>}

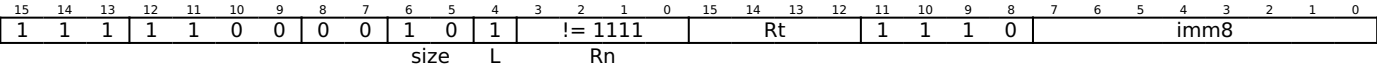
Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `n == t && n != 15`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

```
LDRT{<c>}{<q>}<Rt>, [<Rn> {, #<+><imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> For the "A1" variant: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
For the "A2" and "T1" variants: is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.
- +/- For the "A1" variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":
- | U | +/- |
|---|-----|
| 0 | - |
| 1 | + |
- For the "A2" variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":
- | U | +/- |
|---|-----|
| 0 | - |
| 1 | + |
- <imm> For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
For the "T1" variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.
- <Rm> Is the general-purpose index register, encoded in the "Rm" field.
- <shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).
- +
- Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  if PSTATE.EL == EL2 then UnpredictableProcedure(); end; // Hyp mode
  let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                          else imm32);
  let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
  let address : bits(32) = if postindex then R(n) else offset_addr;
  let data : bits(32) = MemU\_unpriv{32}(address);
  if postindex then R(n) = offset_addr; end;
  R(t) = data;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDR (immediate).

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd					!= 0000				0	0	0	Rm			
cond				opc				S				Rn				imm5				stype											

Encoding for the MOV, shift or rotate by value variant

`LSL{<c>}{<q>} {<Rd>, }<Rm>, #<imm>`

is equivalent to

`MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>`

and is always the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	!= 00000					Rm			Rd		
op					imm5										

Encoding for the T2 variant

`LSL<c>{<q>} {<Rd>, }<Rm>, #<imm>`

is equivalent to

`MOV<c>{<q>} <Rd>, <Rm>, LSL #<imm>`

and is the preferred disassembly when `InITBlock()`.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	0	1	0	0	1	0	0	1	1	1	1	(0)	imm3				Rd				imm2		0	0	Rm			
op1						S		Rn						stype																		

Encoding for the MOV, shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00))`

`LSL{<c>}{<q>} {<Rd>, }<Rm>, #<imm>`

`LSL<c>.W {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)`

is equivalent to

`MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>`

and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 MOV, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC . For the "T2" and "T3 MOV, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.

<Rm>	For the "A1 MOV, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
	For the "T2" and "T3 MOV, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For the "A1 MOV, shift or rotate by value" variant: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32.
	For the "T2" variant: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32.
	For the "T3 MOV, shift or rotate by value" variant: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd				Rs				0	0	0	1	Rm			
cond				opc				S				Rn												stype							

Encoding for the Not flag setting variant

LSL{<c>}{<q>}{<Rd>, }<Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>}<Rd>, <Rm>, LSL <Rs>

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0	Rs			Rdm		
op															

Encoding for the Logical shift left variant

LSL<c>{<q>}{<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOV<c>{<q>}<Rdm>, <Rdm>, LSL <Rs>

and is the preferred disassembly when `InitBlock()`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	0	Rm				1	1	1	1	Rd				0	0	0	0	Rs			
stype												S																			

Encoding for the Not flag setting variant

LSL{<c>}{<q>}{<Rd>, }<Rm>, <Rs>

LSL<c>.W {<Rd>, }<Rm>, <Rs> // (InitBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))

is equivalent to

MOV{<c>}{<q>}<Rd>, <Rm>, LSL <Rs>

and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSLS (immediate)

Logical Shift Left, setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR <current_mode>.
- The PE checks SPSR <current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd				!= 00000				0	0	0	Rm				
cond								opc				S	Rn								imm5				stype						

Encoding for the MOVS, shift or rotate by value variant

LSLS{<c>}{<q>}{<Rd>}, <Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>}{<Rd>}, <Rm>, LSL #<imm>

and is always the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	!= 00000					Rm			Rd		
op					imm5										

Encoding for the T2 variant

LSLS{<q>}{<Rd>}, <Rm>, #<imm>

is equivalent to

MOVS<c>{<q>}{<Rd>}, <Rm>, LSL #<imm>

and is the preferred disassembly when `!InITBlock()`.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	1	(0)	imm3				Rd				imm2				0	0	Rm			
op1							S		Rn														stype											

Encoding for the MOVS, shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00))`

LSLS{<c>}{<q>}{<Rd>}, <Rm>, #<imm>

LSLS.W {<Rd>}, <Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)

is equivalent to

MOVS{<c>}{<q>}{<Rd>}, <Rm>, LSL #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd>	For the "A1 MOVS, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores <i>PSTATE</i> from SPSR_<current_mode>. For the "T2" and "T3 MOVS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1 MOVS, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T2" and "T3 MOVS, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For the "A1 MOVS, shift or rotate by value" variant: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32. For the "T2" variant: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32. For the "T3 MOVS, shift or rotate by value" variant: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSLS (register)

Logical Shift Left, setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd				Rs				0	0	0	1	Rm			
cond				opc				S				Rn												stype							

Encoding for the Flag setting variant

LSLS{<c>}{<q>}{<Rd>, }<Rm>, <Rs>
is equivalent to
MOVS{<c>}{<q>}<Rd>, <Rm>, LSL <Rs>
and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0	Rs			Rdm		
op															

Encoding for the Logical shift left variant

LSLS{<q>}{<Rdm>, }<Rdm>, <Rs>
is equivalent to
MOVS{<q>}<Rdm>, <Rdm>, LSL <Rs>
and is the preferred disassembly when `!InITBlock()`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	1	Rm				1	1	1	1	Rd				0	0	0	0	Rs			
stype												S																			

Encoding for the Flag setting variant

LSLS{<c>}{<q>}{<Rd>, }<Rm>, <Rs>
LSLS.W {<Rd>, }<Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
is equivalent to
MOVS{<c>}{<q>}<Rd>, <Rm>, LSL <Rs>
and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd					imm5					0	1	0	Rm			
cond				opc				S				Rn																stype				

Encoding for the MOV, shift or rotate by value variant

LSR{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>

and is always the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	imm5					Rm			Rd		
op															

Encoding for the T2 variant

LSR<c>{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV<c>{<q>} <Rd>, <Rm>, LSR #<imm>

and is the preferred disassembly when `InITBlock()`.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	0	1	0	0	1	0	0	1	1	1	1	(0)	imm3				Rd				imm2		0	1	Rm			
op1						S		Rn																		stype						

Encoding for the MOV, shift or rotate by value variant

LSR{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

LSR<c>.W {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 MOV, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC . For the "T2" and "T3 MOV, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1 MOV, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T2" and "T3 MOV, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.

<imm>

For the "A1 MOV, shift or rotate by value" and "T2" variants: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For the "T3 MOV, shift or rotate by value" variant: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd				Rs				0	0	1	1	Rm			
cond				opc				S				Rn												stype							

Encoding for the Not flag setting variant

LSR{<c>}{<q>}{<Rd>, }<Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>}<Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Rs			Rdm		
op															

Encoding for the Logical shift right variant

LSR<c>{<q>}{<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOV<c>{<q>}<Rdm>, <Rdm>, LSR <Rs>

and is the preferred disassembly when `InitBlock()`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	0	0	0	1	0	Rm				1	1	1	1	Rd				0	0	0	0	Rs							
stype																S																			

Encoding for the Not flag setting variant

LSR{<c>}{<q>}{<Rd>, }<Rm>, <Rs>

LSR<c>.W {<Rd>, }<Rm>, <Rs> // (InitBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))

is equivalent to

MOV{<c>}{<q>}<Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSRS (immediate)

Logical Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd					imm5					0	1	0	Rm			
cond				opc				S				Rn								stype												

Encoding for the MOVS, shift or rotate by value variant

LSRS{<c>}{<q>}{<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>}{<Rd>, }<Rm>, LSR #<imm>

and is always the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	imm5					Rm			Rd		
op															

Encoding for the T2 variant

LSRS{<q>}{<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS<c>{<q>}{<Rd>, }<Rm>, LSR #<imm>

and is the preferred disassembly when `!InITBlock()`.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	1	(0)	imm3			Rd			imm2			0	1	Rm				
op1							S					Rn					stype															

Encoding for the MOVS, shift or rotate by value variant

LSRS{<c>}{<q>}{<Rd>, }<Rm>, #<imm>

LSRS.W {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)

is equivalent to

MOVS{<c>}{<q>}{<Rd>, }<Rm>, LSR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd>	For the "A1 MOVS, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores <i>PSTATE</i> from SPSR_<current_mode>. For the "T2" and "T3 MOVS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1 MOVS, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T2" and "T3 MOVS, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For the "A1 MOVS, shift or rotate by value" and "T2" variants: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32. For the "T3 MOVS, shift or rotate by value" variant: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSRS (register)

Logical Shift Right, setting flags (register) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd					Rs			0	0	1	1	Rm			
cond				opc				S				Rn				styp															

Encoding for the Flag setting variant

LSRS{<c>}{<q>}{<Rd>, <Rm>, <Rs>}

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Rs			Rdm		
op															

Encoding for the Logical shift right variant

LSRS{<q>}{<Rdm>, <Rdm>, <Rs>}

is equivalent to

MOVS{<q>}<Rdm>, <Rdm>, LSR <Rs>

and is the preferred disassembly when `!InITBlock()`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	1	Rm			1	1	1	1	Rd			0	0	0	0	Rs					
styp																S															

Encoding for the Flag setting variant

LSRS{<c>}{<q>}{<Rd>, <Rm>, <Rs>}

LSRS.W {<Rd>, <Rm>, <Rs>} // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MCR

Move to System register from general-purpose register or execute a System instruction. This instruction copies the value of a general-purpose register to a System register, or executes a System instruction.

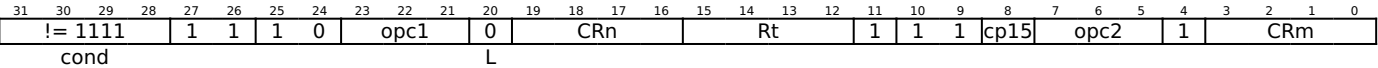
The System register and System instruction descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see *About the AArch32 System register interface* and *General behavior of System registers*.

In an implementation that includes EL2, MCR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *EL2 configurable instruction enables, disables, and traps*.

Because of the range of possible traps to Hyp mode, the MCR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



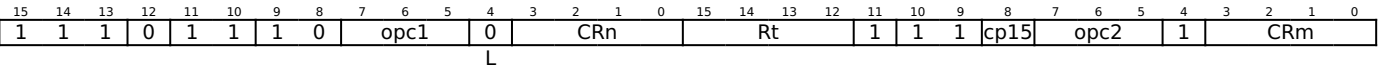
Encoding

MCR{<c>}{<q>}<coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

MCR{<c>}{<q>}<coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c>

See *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15
- <opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc1" field.
- <Rt>

Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <CRn>

Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRn" field.
- <CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.
- <opc2>

Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc2" field.
- The possible values of { <coproc>, <opc1>, <CRn>, <CRm>, <opc2> }

encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_SysRegWrite(cp, ThisInstr(), t);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MCRR

Move to System register from two general-purpose registers. This instruction copies the values of two general-purpose registers to a System register.

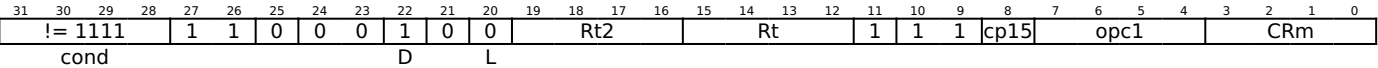
The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MCRR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCRR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [EL2 configurable instruction enables, disables, and traps](#).

Because of the range of possible traps to Hyp mode, the MCRR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



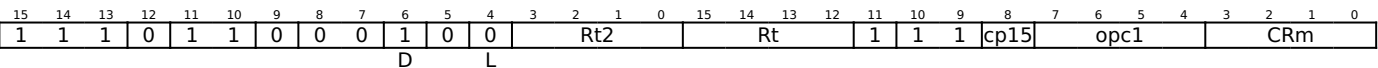
Encoding

MCRR{<c>}{<q>}<coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

T1



Encoding

MCRR{<c>}{<q>}<coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <coproc> Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15
- <opc1> Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.
- <Rt> Is the first general-purpose register that is transferred into, encoded in the "Rt" field.
- <Rt2> Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.
- <CRm> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of { <coproc>, <opc1>, <CRm> } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <Rt2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_SysRegWrite64(cp, ThisInstr(), t, t2);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLA, MLAS

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values. In an A32 instruction, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	0	1	S	Rd				Ra				Rm				1	0	0	1	Rn			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

MLAS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the Not flag setting variant

Applies when (S == 0)

MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let setflags : boolean = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	0	0	0	Rn				!= 1111				Rd				0	0	0	0	Rm			
op1									Ra									op2													

Encoding

MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```
if Ra == '1111' then See("MUL"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let setflags : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : integer = SInt(R(n)); // operand1 = UInt(R[n]) produces the same final results
    let operand2 : integer = SInt(R(m)); // operand2 = UInt(R[m]) produces the same final results
    let addend : integer = SInt(R(a)); // addend = UInt(R[a]) produces the same final results
    let result : integer = operand1 * operand2 + addend;
    R(d) = result[31:0];
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result[31:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLS

Multiply and Subtract multiplies two register values, and subtracts the product from a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	1	1	0	Rd				Ra				Rm				1	0	0	1	Rn			
cond				opc								S																			

Encoding

MLS{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	0	0	0	Rn				Ra				Rd				0	0	0	1	Rm			
op1																												op2			

Encoding

MLS{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Ra> Is the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : integer = SInt(R(n)); // operand1 = UInt(R[n]) produces the same final results
    let operand2 : integer = SInt(R(m)); // operand2 = UInt(R[m]) produces the same final results
    let addend : integer = SInt(R(a)); // addend = UInt(R[a]) produces the same final results
    let result : integer = addend - operand1 * operand2;
    R(d) = result[31:0];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

MOV, MOVS (immediate)

Move (immediate) writes an immediate value to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The MOV variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	1	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd					imm12											
cond				opc				Rn																								

Encoding for the MOV variant

Applies when (S == 0)

MOV{<c>}{<q>} <Rd>, #<const>

Encoding for the MOVS variant

Applies when (S == 1)

MOVS{<c>}{<q>} <Rd>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let (imm32, carry) : (bits(32), bit) = A32ExpandImm_C(imm12, PSTATE.C);
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	1	1	0	0	0	0	imm4				Rd					imm12											
cond				H																												

Encoding

MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can not be represented in A1)

MOVW{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in A1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend()(imm4::imm12);
let carry : bit = ARBITRARY : bit;
if d == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							
op															

Encoding

MOV<c>{<q>} <Rd>, #<imm8> // (InITBlock())

MOVS{<q>} <Rd>, #<imm8> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend()(imm8);
let carry : bit = PSTATE.C;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																																																
1				1				1				0				i				0				0				0				1				0				S				1				1				1				1				0				imm3				Rd				imm8							
op1												Rn																																																																			

Encoding for the MOV variant

Applies when (S == 0)

```
MOV{<c>}{<q>} <Rd>, #<const>
```

```
MOV<c>.W <Rd>, #<const> // (Inside IT block, and <Rd>, <const> can be represented in T1)
```

Encoding for the MOVS variant

Applies when (S == 1)

```
MOVS{<c>}{<q>} <Rd>, #<const>
```

```
MOVS.W <Rd>, #<const> // (Outside IT block, and <Rd>, <const> can be represented in T1)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let (imm32, carry) : (bits(32), bit) = T32ExpandImm_C(i::imm3::imm8, PSTATE.C);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	1	0	0	1	0	0	imm4				0	imm3				Rd				imm8							
o1																																

Encoding

```
MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> cannot be represented in T1 or T2)
```

```
MOVW{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in T1 or T2)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{(imm4::i::imm3::imm8)};
let carry : bit = ARBITRARY : bit;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 MOV" and "A1 MOVS" variants: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used: - For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. For the "A2", "T1", "T2 MOV", "T2 MOVS", and "T3" variants: is the general-purpose destination register, encoded in the "Rd" field.
<const>	For the "A1 MOV" and "A1 MOVS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values. For the "T2 MOV" and "T2 MOVS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.
<imm16>	For the "A2" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. For the "T3" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.
<imm8>	Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : bits(32) = imm32;
    if d == 15 then           // Can only occur for encoding A1
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV, MOVS (register)

Move (register) copies a value from a register to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The MOV variant of the instruction is a branch. In the T32 instruction set (encoding T1) this is a simple branch, and in the A32 instruction set it is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is used by the aliases [ASRS \(immediate\)](#), [ASR \(immediate\)](#), [LSLS \(immediate\)](#), [LSL \(immediate\)](#), [LSRS \(immediate\)](#), [LSR \(immediate\)](#), [RORS \(immediate\)](#), [ROR \(immediate\)](#), [RRXS](#), and [RRX](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!	=	1	1	1	1	0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				imm5				stype	0	Rm			
cond				opc				Rn																							

Encoding for the MOV, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

MOV{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MOV, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

Encoding for the MOVS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

MOVS{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MOVS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		
op															

Encoding

MOV{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(D::Rd);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if d == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	!= 11		imm5					Rm			Rd		
op															

Encoding

```
MOV<C>{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (InITBlock())
MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(op, imm5);
if op == '00' && imm5 == '00000' && InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `op == '00' && imm5 == '00000' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passed its condition code check.
- The instruction executes as NOP, as if it failed its condition code check.
- The instruction executes as MOV Rd, Rm.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	imm3				Rd				imm2		stype		Rm			
op1												Rn																				

Encoding for the MOV, rotate right with extend variant

Applies when `(S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

Encoding for the MOV, shift or rotate by value variant

Applies when `(S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
MOV{<c>}.W <Rd>, <Rm> {, LSL #0} // (<Rd>, <Rm> can be represented in T1)
MOV<C>.W <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block, and <Rd>, <Rm>, <shift>, <amount> can be represented in T2)
```

Encoding for the MOVS, rotate right with extend variant

Applies when `(S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)`

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

Encoding for the MOVS, shift or rotate by value variant

Applies when `(S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))`

```
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
MOVS.W <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block, and <Rd>, <Rm>, <shift>, <amount> can be represented in T1 or T2)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).
 <q> See [Standard assembler syntax fields](#).

<Rd> For the "A1 MOV, rotate right with extend", "A1 MOV, shift or rotate by value", "A1 MOVS, rotate right with extend", and "A1 MOVS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. Arm deprecates use of the instruction if <Rn> is the PC.
- For the MOVS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. Arm deprecates use of the instruction if <Rn> is not the LR, or if the optional shift or RRX argument is specified.

For the "T1" variant: is the general-purpose destination register, encoded in the "D:Rd" field. If the PC is used:

- The instruction causes a branch to the address moved to the PC. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The instruction must either be outside an IT block or the last instruction of an IT block.

For the "T2", "T3 MOV, rotate right with extend", "T3 MOV, shift or rotate by value", "T3 MOVS, rotate right with extend", and "T3 MOVS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For the "A1 MOV, rotate right with extend", "A1 MOV, shift or rotate by value", "A1 MOVS, rotate right with extend", "A1 MOVS, shift or rotate by value", and "T1" variants: is the general-purpose source register, encoded in the "Rm" field. The PC can be used. Arm deprecates use of the instruction if <Rd> is the PC.

For the "T2", "T3 MOV, rotate right with extend", "T3 MOV, shift or rotate by value", "T3 MOVS, rotate right with extend", and "T3 MOVS, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.

<shift> For the "A1 MOV, shift or rotate by value", "A1 MOVS, shift or rotate by value", "T3 MOV, shift or rotate by value", and "T3 MOVS, shift or rotate by value" variants: is the type of shift to be applied to the source register, encoded in "stype":

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

For the "T2" variant: is the type of shift to be applied to the source register, encoded in "op":

op	<shift>
00	LSL
01	LSR
10	ASR

<amount> For the "A1 MOV, shift or rotate by value" and "A1 MOVS, shift or rotate by value" variants: is the shift amount, in the range 0 to 31 (when <shift> = LSL), or 1 to 31 (when <shift> = ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For the "T2" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For the "T3 MOV, shift or rotate by value" and "T3 MOVS, shift or rotate by value" variants: is the shift amount, in the range 0 to 31 (when <shift> = LSL) or 1 to 31 (when <shift> = ROR), or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Alias Conditions

Alias	Of variant	Is preferred when
ASRS (immediate)	T3 MOVS, shift or rotate by value, A1 MOVS, shift or rotate by value	<code>S == '1' && stype == '10'</code>
ASRS (immediate)	T2	<code>op == '10' && !InITBlock()</code>

Alias	Of variant	Is preferred when
ASR (immediate)	T3 MOV, shift or rotate by value, A1 MOV, shift or rotate by value	<code>S == '0' && stype == '10'</code>
ASR (immediate)	T2	<code>op == '10' && InITBlock()</code>
LSLS (immediate)	A1 MOVS, shift or rotate by value	<code>S == '1' && imm5 != '00000' && stype == '00'</code>
LSLS (immediate)	T2	<code>op == '00' && imm5 != '00000' && !InITBlock()</code>
LSLS (immediate)	T3 MOVS, shift or rotate by value	<code>(S == '1' && stype == '00' && !(imm3 == '000')) (S == '1' && stype == '00' && !(imm2 == '00'))</code>
LSL (immediate)	A1 MOV, shift or rotate by value	<code>S == '0' && imm5 != '00000' && stype == '00'</code>
LSL (immediate)	T2	<code>op == '00' && imm5 != '00000' && InITBlock()</code>
LSL (immediate)	T3 MOV, shift or rotate by value	<code>(S == '0' && stype == '00' && !(imm3 == '000')) (S == '0' && stype == '00' && !(imm2 == '00'))</code>
LSRS (immediate)	T3 MOVS, shift or rotate by value, A1 MOVS, shift or rotate by value	<code>S == '1' && stype == '01'</code>
LSRS (immediate)	T2	<code>op == '01' && !InITBlock()</code>
LSR (immediate)	T3 MOV, shift or rotate by value, A1 MOV, shift or rotate by value	<code>S == '0' && stype == '01'</code>

Alias	Of variant	Is preferred when
LSR (immediate)	T2	<code>op == '01' && InITBlock()</code>
RORS (immediate)	A1 MOVS, shift or rotate by value	<code>S == '1' && imm5 != '00000' && stype == '11'</code>
RORS (immediate)	T3 MOVS, shift or rotate by value	<code>(S == '1' && stype == '11' && !(imm3 == '000')) (S == '1' && stype == '11' && !(imm2 == '00'))</code>
ROR (immediate)	A1 MOV, shift or rotate by value	<code>S == '0' && imm5 != '00000' && stype == '11'</code>
ROR (immediate)	T3 MOV, shift or rotate by value	<code>(S == '0' && stype == '11' && !(imm3 == '000')) (S == '0' && stype == '11' && !(imm2 == '00'))</code>
RRXS	A1 MOVS, rotate right with extend	<code>S == '1' && imm5 == '00000' && stype == '11'</code>
RRXS	T3 MOVS, rotate right with extend	<code>S == '1' && imm3 == '000' && imm2 == '00' && stype == '11'</code>
RRX	A1 MOV, rotate right with extend	<code>S == '0' && imm5 == '00000' && stype == '11'</code>
RRX	T3 MOV, rotate right with extend	<code>S == '0' && imm3 == '000' && imm2 == '00' && stype == '11'</code>

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = shifted;
    if d == 15 then
        if setflags then
            ALUEXCEPTIONReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;

```


Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV, MOVS (register-shifted register)

Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases [ASRS \(register\)](#), [ASR \(register\)](#), [LSLS \(register\)](#), [LSL \(register\)](#), [LSRS \(register\)](#), [LSR \(register\)](#), [RORS \(register\)](#), and [ROR \(register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				Rs				0	stype	1	Rm				
cond								opc				Rn																			

Encoding for the Flag setting variant

Applies when (S == 1)

```
MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when (S == 0)

```
MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	x	x	x	Rs			Rdm		
						op									

Encoding for the Arithmetic shift right variant

Applies when (op == 0100)

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs> // (InITBlock())
```

```
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs> // (Outside IT block)
```

Encoding for the Logical shift left variant

Applies when (op == 0010)

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs> // (InITBlock())
```

```
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs> // (Outside IT block)
```

Encoding for the Logical shift right variant

Applies when (op == 0011)

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs> // (InITBlock())
```

```
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs> // (Outside IT block)
```

Encoding for the Rotate right variant

Applies when (op == 0111)

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs> // (InITBlock())
```

```
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs> // (Outside IT block)
```

Decode for all variants of this encoding

```
if ! op IN {'0010', '0011', '0100', '0111'} then See("Related encodings"); end;
let d : integer = UInt(Rdm);
let m : integer = UInt(Rdm);
let s : integer = UInt(Rs);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = DecodeRegShift(op[2]::op[0]);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	styp	e	S	Rm				1	1	1	1	Rd				0	0	0	0	Rs			

Encoding for the Flag setting variant

Applies when (S == 1)

```
MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

MOVS.W <Rd>, <Rm>, <shift> <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
```

Encoding for the Not flag setting variant

Applies when (S == 0)

```
MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

MOV<c>.W <Rd>, <Rm>, <shift> <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SType = DecodeRegShift(stype);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

Related encodings: In encoding T1, for an op field value that is not described above, see [Data-processing \(two low registers\)](#).
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rm>	Is the general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<Rs>	Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.										
<Rdm>	Is the general-purpose source register and the destination register, encoded in the "Rdm" field.										

Alias Conditions

Alias	Of variant	Is preferred when
ASRS (register)	A1 Flag setting	S == '1' && stype == '10'
ASRS (register)	T1 Arithmetic shift right	op == '0100' && !InITBlock()
ASRS (register)	T2 Flag setting	stype == '10' && S == '1'
ASR (register)	A1 Not flag setting	S == '0' && stype == '10'
ASR (register)	T1 Arithmetic shift right	op == '0100' && InITBlock()
ASR (register)	T2 Not flag setting	stype == '10' && S == '0'
LSLS (register)	A1 Flag setting	S == '1' && stype == '00'
LSLS (register)	T1 Logical shift left	op == '0010' && !InITBlock()
LSLS (register)	T2 Flag setting	stype == '00' && S == '1'
LSL (register)	A1 Not flag setting	S == '0' && stype == '00'
LSL (register)	T1 Logical shift left	op == '0010' && InITBlock()
LSL (register)	T2 Not flag setting	stype == '00' && S == '0'
LSRS (register)	A1 Flag setting	S == '1' && stype == '01'
LSRS (register)	T1 Logical shift right	op == '0011' && !InITBlock()

Alias	Of variant	Is preferred when
LSRS (register)	T2 Flag setting	<code>stype == '01' && S == '1'</code>
LSR (register)	A1 Not flag setting	<code>S == '0' && stype == '01'</code>
LSR (register)	T1 Logical shift right	<code>op == '0011' && InITBlock()</code>
LSR (register)	T2 Not flag setting	<code>stype == '01' && S == '0'</code>
RORS (register)	A1 Flag setting	<code>S == '1' && stype == '11'</code>
RORS (register)	T1 Rotate right	<code>op == '0111' && !InITBlock()</code>
RORS (register)	T2 Flag setting	<code>stype == '11' && S == '1'</code>
ROR (register)	A1 Not flag setting	<code>S == '0' && stype == '11'</code>
ROR (register)	T1 Rotate right	<code>op == '0111' && InITBlock()</code>
ROR (register)	T2 Not flag setting	<code>stype == '11' && S == '0'</code>

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (result, carry) : (bits(32), bit) = Shift\_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

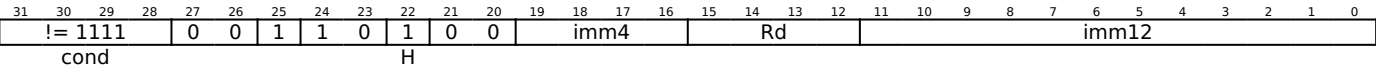
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV_T

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



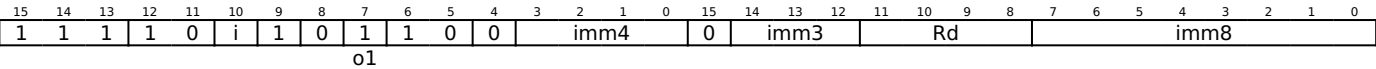
Encoding

MOV_T{<c>}{<q>} <Rd>, #<imm16>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm16 : bits(16) = imm4::imm12;
if d == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

MOV_T{<c>}{<q>} <Rd>, #<imm16>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm16 : bits(16) = imm4::i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <imm16> For the "A1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.
For the "T1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[31:16] = imm16;
    // R[d]<15:0> unchanged
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

MRC

Move to general-purpose register from System register. This instruction copies the value of a System register to a general-purpose register. The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MRC accesses to system control registers can be trapped to Hyp mode, meaning that an attempt to execute an MRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [EL2 configurable instruction enables, disables, and traps](#).

Because of the range of possible traps to Hyp mode, the MRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
!= 1111				1	1	1	0	opc1				1	CRn				Rt				1	1	1	cp15	opc2			1	CRm							
cond											L																									

Encoding

MRC{<c>}{<q>}<coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1			1	CRn			Rt				1	1	1	cp15	opc2			1	CRm				
											L																				

Encoding

MRC{<c>}{<q>}<coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

Assembler Symbols

- <c>

See [Standard assembler syntax fields](#).
- <q>

See [Standard assembler syntax fields](#).
- <coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15
- <opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc1" field.
- <Rt>

Is the general-purpose register to be transferred or APSR_nzcv (encoded as 0b1111), encoded in the "Rt" field. If APSR_nzcv is used, bits [31:28] of the transferred value are written to the [PSTATE](#) condition flags.
- <CRn>

Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRn" field.
- <CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.
- <opc2>

Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc2" field.
- The possible values of { <coproc>, <opc1>, <CRn>, <CRm>, <opc2> }

encode the entire System register and System instruction encoding space.

Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    if t != 15 || AArch32\_SysRegReadCanWriteAPSR(cp, ThisInstr()) then
        AArch32_SysRegRead(cp, ThisInstr(), t);
    else
        UnpredictableProcedure();
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MRRC

Move to two general-purpose registers from System register. This instruction copies the value of a System register to two general-purpose registers. The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MRRC accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MRRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [EL2 configurable instruction enables, disables, and traps](#).

Because of the range of possible traps to Hyp mode, the MRRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	0	0	1	0	1	Rt2				Rt				1	1	1	cp15	opc1				CRm			
cond								D				L																			

Encoding

MRRC{<c>}{<q> <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	1	0	1	Rt2				Rt				1	1	1	cp15	opc1				CRm			
								D				L																			

Encoding

MRRC{<c>}{<q> <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15

<opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.

<Rt>

Is the first general-purpose register that is transferred into, encoded in the "Rt" field.

<Rt2>

Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.

<CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of { <coproc>, <opc1>, <CRm> } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <Rt2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_SysRegRead64(cp, ThisInstr(), t, t2);
end;
```

MRS

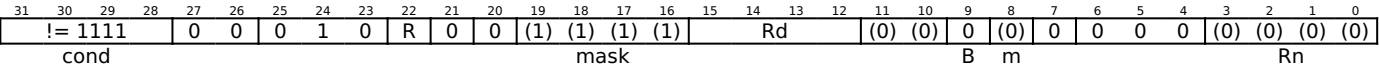
Move Special register to general-purpose register moves the value of the [APSR](#), [CPSR](#), or [SPSR](#) `<current_mode>` into a general-purpose register. Arm recommends the APSR form when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see [APSR](#).

An MRS that accesses the [SPSRs](#) is UNPREDICTABLE if executed in User mode or System mode.

An MRS that is executed in User mode and accesses the [CPSR](#) returns an UNKNOWN value for the [CPSR](#).{E, A, I, F, M} fields.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



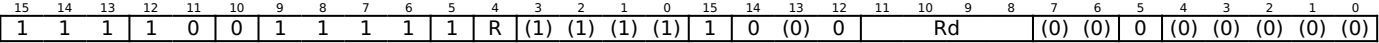
Encoding

```
MRS{<c>}{<q>} <Rd>, <spec_reg>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
if d == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
MRS{<c>}{<q>} <Rd>, <spec_reg>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<spec_reg>` Is the special register to be accessed, encoded in "R":

R	<spec_reg>
0	CPSR APSR
1	SPSR

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if read_spsr then
        if PSTATE.M IN {M32\_User,M32\_System} then
            UnpredictableProcedure();
        else
            R(d) = SPSR\_curr();
        end;
    else
        // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
        let maskval : bits(32) = '11111000 11101111 00000011 11011111';
        var psr_val : bits(32) = GetPSRFromPSTATE{32}(AArch32\_NonDebugState) AND maskval;
        if PSTATE.EL == EL0 then
            // If accessed from User mode return UNKNOWN values for E, A, I, F bits, bits<9:6>,
            // and for the M field, bits<4:0>
            psr_val[22] = ARBITRARY : bits(1);
            psr_val[9:6] = ARBITRARY : bits(4);
            psr_val[4:0] = ARBITRARY : bits(5);
        end;
        R(d) = psr_val;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If [PSTATE.M](#) IN {[M32_User](#), [M32_System](#)} && [read_spsr](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MRS (Banked register)

Move to Register from Banked or Special register moves the value from the Banked general-purpose register or *Saved Program Status Registers (SPSRs)* of the specified mode, or the value of *ELR_hyp*, to a general-purpose register.

MRS (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MRS (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MRS (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the Banked register transfer instructions*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	R	0	0	M1			Rd			(0) (0)		1		M	0	0	0	0	(0)	(0)	(0)	(0)	
cond				B																						Rn					

Encoding

MRS{<c>}{<q>} <Rd>, <banked_reg>

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
if d == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	R	M1			1	0	(0)	0	Rd			(0)	(0)	1	M	(0)	(0)	(0)	(0)		

Encoding

MRS{<c>}{<q>} <Rd>, <banked_reg>

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<banked_reg>

Is the name of the banked register to be transferred to or from, encoded in “(R :: M :: M1)”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE


```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        let c : Constraint = ConstrainUnpredictable(Unpredictable BankedRegister);
        case c of
            when Constraint UNDEF =>
                Undefined();
            when Constraint NOP =>
                ExecuteAsNOP();
            when Constraint UNKNOWN =>
                R(d) = ARBITRARY : bits(32);
        end;
    else
        let mode : bits(5) = PSTATE.M;
        if read_spsr then
            // Check for CONSTRAINED UNPREDICTABLE cases
            var valid : boolean;
            var c : Constraint = Constraint NONE;
            (valid, c) = SPSRaccessValid(SYSm, mode);

            if valid then
                case SYSm of
                    when '01110' => R(d) = SPSR_fiq() [31:0];
                    when '10000' => R(d) = SPSR_irq() [31:0];
                    when '10010' => R(d) = SPSR_svc() [31:0];
                    when '10100' => R(d) = SPSR_abt() [31:0];
                    when '10110' => R(d) = SPSR_und() [31:0];
                    when '11100' =>
                        if !ELUsingAArch32(EL3) then AArch64_MonitorModeTrap(); end;
                        R(d) = SPSR_mon();
                    when '11110' => R(d) = SPSR_hyp() [31:0];
                end;
            else
                case c of
                    when Constraint UNDEF =>
                        Undefined();
                    when Constraint NOP =>
                        ExecuteAsNOP();
                    when Constraint UNKNOWN =>
                        R(d) = ARBITRARY : bits(32);
                    when Constraint ANYREG =>
                        R(d) = ReadAnyAllocatedSPSR();
                end;
            end;
        else
            // Check for CONSTRAINED UNPREDICTABLE cases
            var valid : boolean;
            var c : Constraint = Constraint NONE;
            (valid, c) = BankedRegisterAccessValid(SYSm, mode);

            if valid then
                case SYSm of
                    when '00xxx' =>
                        // Access the User mode registers
                        let m : integer = UInt(SYSm[2:0]) + 8;
                        R(d) = Rmode(m, M32_User);
                    when '01xxx' =>
                        // Access the FIQ mode registers
                        let m : integer = UInt(SYSm[2:0]) + 8;
                        R(d) = Rmode(m, M32_FIQ);
                    when '1000x' =>
                        // Access the IRQ mode registers
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        R(d) = Rmode(m, M32_IRQ);
                    when '1001x' =>
                        // Access the Supervisor mode registers
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        R(d) = Rmode(m, M32_Svc);
                    when '1010x' =>
                        // Access the Abort mode registers
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        R(d) = Rmode(m, M32_Abort);
                    when '1011x' =>
                        // Access the Undefined mode registers
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        R(d) = Rmode(m, M32_Undef);
                    when '1110x' =>
                        // Access Monitor registers
                        if !ELUsingAArch32(EL3) then AArch64_MonitorModeTrap(); end;
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        R(d) = Rmode(m, M32_Monitor);
                    when '11110' =>
                        // Access ELR_hyp register
                        R(d) = ELR_hyp();
                    when '11111' =>
                        // Access SP_hyp register
                        R(d) = Rmode(13, M32_Hyp);
                end;
            else
                case c of
                    when Constraint UNDEF =>
                        Undefined();
                    when Constraint NOP =>

```

```

        ExecuteAsNOP();
    when Constraint UNKNOWN =>
        R(d) = ARBITRARY :bits(32);
    when Constraint ANYREG =>
        R(d) = ReadAnyAllocatedRegister();
    end;
end;
end;
end;
end;

```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSR (Banked register)

Move to Banked or Special register from general-purpose register moves the value of a general-purpose register to the Banked general-purpose register or *Saved Program Status Registers (SPSRs)* of the specified mode, or to *ELR_hyp*.

MSR (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MSR (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MSR (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the Banked register transfer instructions*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	R	1	0	M1				(1)	(1)	(1)	(1)	(0)	(0)	1	M	0	0	0	0	Rn			
cond				Rd												B															

Encoding

MSR{<c>}{<q>} <banked_reg>, <Rn>

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
if n == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	R	Rn				1	0	(0)	0	M1				(0)	(0)	1	M	(0)	(0)	(0)	(0)

Encoding

MSR{<c>}{<q>} <banked_reg>, <Rn>

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

<banked_reg>

Is the name of the banked register to be transferred to or from, encoded in “(R :: M :: M1)”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

<Rn>

Is the general-purpose source register, encoded in the "Rn" field.


```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        let c : Constraint = ConstrainUnpredictable(Unpredictable\_BankedRegister);
        case c of
            when Constraint\_UNDEF =>
                Undefined();
            when Constraint\_NOP, Constraint\_UNKNOWNN =>
                ExecuteAsNOP();
        end;
    else
        let mode : bits(5) = PSTATE.M;
        if write_spsr then
            // Check for CONSTRAINED UNPREDICTABLE cases
            var valid : boolean;
            var c : Constraint = Constraint\_NONE;
            (valid, c) = SPSRaccessValid(SYSm, mode);

            if valid then
                case SYSm of
                    when '01110' => SPSR\_fiq() [31:0] = R(n);
                    when '10000' => SPSR\_irq() [31:0] = R(n);
                    when '10010' => SPSR\_svc() [31:0] = R(n);
                    when '10100' => SPSR\_abt() [31:0] = R(n);
                    when '10110' => SPSR\_und() [31:0] = R(n);
                    when '11100' =>
                        if !ELUsingAArch32(EL3) then AArch64\_MonitorModeTrap(); end;
                        SPSR\_mon() [31:0] = R(n);
                    when '11110' => SPSR\_hyp() [31:0] = R(n);
                end;
            else
                case c of
                    when Constraint\_UNDEF =>
                        Undefined();
                    when Constraint\_NOP, Constraint\_UNKNOWNN =>
                        ExecuteAsNOP();
                    when Constraint\_ANYREG =>
                        WriteAnyAllocatedSPSR(R(n));
                end;
            end;
        else
            // Check for CONSTRAINED UNPREDICTABLE cases
            var valid : boolean;
            var c : Constraint = Constraint\_NONE;
            (valid, c) = BankedRegisterAccessValid(SYSm, mode);

            if valid then
                case SYSm of
                    when '00xxx' => // Access the User mode registers
                        let m : integer = UInt(SYSm[2:0]) + 8;
                        Rmode(m, M32\_User) = R(n);
                    when '01xxx' => // Access the FIQ mode registers
                        let m : integer = UInt(SYSm[2:0]) + 8;
                        Rmode(m, M32\_FIQ) = R(n);
                    when '1000x' => // Access the IRQ mode registers
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        Rmode(m, M32\_IRQ) = R(n);
                    when '1001x' => // Access the Supervisor mode registers
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        Rmode(m, M32\_Svc) = R(n);
                    when '1010x' => // Access the Abort mode registers
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        Rmode(m, M32\_Abort) = R(n);
                    when '1011x' => // Access the Undefined mode registers
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        Rmode(m, M32\_Undef) = R(n);
                    when '1110x' => // Access Monitor registers
                        if !ELUsingAArch32(EL3) then AArch64\_MonitorModeTrap(); end;
                        // LR when SYSm[0] == 0, otherwise SP
                        let m : integer = 14 - UInt(SYSm[0]);
                        Rmode(m, M32\_Monitor) = R(n);
                    when '11110' => // Access ELR_hyp register
                        ELR\_hyp() = R(n);
                    when '11111' => // Access SP_hyp register
                        Rmode(13, M32\_Hyp) = R(n);
                end;
            else
                case c of
                    when Constraint\_UNDEF =>
                        Undefined();
                    when Constraint\_NOP, Constraint\_UNKNOWNN =>
                        ExecuteAsNOP();
                    when Constraint\_ANYREG =>
                        WriteAnyAllocatedRegister(R(n));
                end;
            end;
        end;
    end;
end;

```

```
        end;  
    end;  
end;  
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSR (immediate)

Move immediate value to Special register moves selected bits of an immediate value to the corresponding bits in the *APSR*, *CPSR*, or *SPSR* <current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, the immediate form of *MSR* is normally only useful at the Application level for writing to *APSR_nzcvq* (*CPSR_f*).

If an *MSR* (immediate) moves selected bits of an immediate value to the *CPSR*, the PE checks whether the value being written to *PSTATE.M* is legal. See *Illegal changes to PSTATE.M*.

An *MSR* (immediate) executed in User mode:

- Is *CONSTRAINED UNPREDICTABLE* if it attempts to update the *SPSR*.
- Otherwise, does not update any *CPSR* field that is accessible only at EL1 or higher,

An *MSR* (immediate) executed in System mode is *CONSTRAINED UNPREDICTABLE* if it attempts to update the *SPSR*.

The *CPSR.E* bit is writable from any mode using an *MSR* instruction. Arm deprecates using this to change its value.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	R	1	0	mask				(1)	(1)	(1)	(1)	imm12											
cond																															

Encoding

Applies when (! (R == 0 && mask == 0000))

MSR{<c>}{<q>} <spec_reg>, #<imm>

Decode for this encoding

```
if mask == '0000' && R == '0' then See("Related encodings"); end;
let imm32 : bits(32) = A32ExpandImm(imm12);
let write_spsr : boolean = (R == '1');
if mask == '0000' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If mask == '0000' && R == '1', then one of the following behaviors must occur:

- The instruction is *UNDEFINED*.
- The instruction executes as *NOP*.

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: *Move Special Register and Hints (immediate)*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<spec_reg> Is one of:

- *APSR* <bits>.
- *CPSR* <fields>.
- *SPSR* <fields>.

For *CPSR* and *SPSR*, <fields> is a sequence of one or more of the following:

c	mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.
x	mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.
s	mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.
f	mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For *APSR*, <bits> is one of *nzcvq*, *g*, or *nzcvqg*. These map to the following *CPSR* <fields> values:

- *APSR_nzcvq* is the same as *CPSR_f* (mask == '1000').
- *APSR_g* is the same as *CPSR_s* (mask == '0100').
- *APSR_nzcvqg* is the same as *CPSR_fs* (mask == '1100').

Arm recommends the *APSR* <bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *The Application Program Status Register, APSR*.

<imm> Is an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32\_User,M32\_System} then
            UnpredictableProcedure();
        else
            SPSRWriteByInstr(imm32, mask);
        end;
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(imm32, mask);
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M` IN {`M32_User`,`M32_System`} && `write_spsr`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSR (register)

Move general-purpose register to Special register moves selected bits of a general-purpose register to the *APSR*, *CPSR* or *SPSR* *<current_mode>*. Because of the Do-Not-Modify nature of its reserved bits, a read-modify-write sequence is normally required when the *MSR* instruction is being used at Application level and its destination is not *APSR_nzcvq* (*CPSR_f*). If an *MSR* (register) moves selected bits of an immediate value to the *CPSR*, the PE checks whether the value being written to *PSTATE.M* is legal. See *Illegal changes to PSTATE.M*. An *MSR* (register) executed in User mode:

- Is UNPREDICTABLE if it attempts to update the *SPSR*.
- Otherwise, does not update any *CPSR* field that is accessible only at EL1 or higher.

An *MSR* (register) executed in System mode is UNPREDICTABLE if it attempts to update the *SPSR*. The *CPSR.E* bit is writable from any mode using an *MSR* instruction. Arm deprecates using this to change its value. It has encodings from the following instruction sets: A32 (*A1*) and T32 (*T1*) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	R	1	0	mask				(1)	(1)	(1)	(1)	(0)	(0)	0	(0)	0	0	0	0	Rn			
cond				Rd												B		m													

Encoding

MSR{<c>}{<q>} <spec_reg>, <Rn>

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
if mask == '0000' then UnpredictableProcedure(); end;
if n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If mask == '0000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	R	Rn				1	0	(0)	0	mask				(0)	(0)	0	(0)	(0)	(0)	(0)	(0)

Encoding

MSR{<c>}{<q>} <spec_reg>, <Rn>

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
if mask == '0000' then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If mask == '0000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

<spec_reg>

Is one of:

- APSR_<bits>.
- CPSR_<fields>.
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

c	mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.
x	mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.
s	mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.
f	mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of nzcvcq, g, or nzcvcqg. These map to the following CPSR_<fields> values:

- APSR_nzcvcq is the same as CPSR_f (mask == '1000').
- APSR_g is the same as CPSR_s (mask == '0100').
- APSR_nzcvcqg is the same as CPSR_fs (mask == '1100').

Arm recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see [The Application Program Status Register, APSR](#).

<Rn>

Is the general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User, M32_System} then
            UnpredictableProcedure();
        else
            SPSRWriteByInstr(R(n), mask);
        end;
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(R(n), mask);
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `write_spsr` && `PSTATE.M IN {M32_User, M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MUL, MULS

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Optionally, it can update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	0	0	S	Rd				(0)	(0)	(0)	(0)	Rm				1	0	0	1	Rn			
cond				opc								RdLo																			

Encoding for the Flag setting variant

Applies when (S == 1)

MULS{<c>}{<q>} <Rd>, <Rn>{, <Rm>}

Encoding for the Not flag setting variant

Applies when (S == 0)

MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	1	Rn			Rdm		
op															

Encoding

MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>} // (InITBlock())

MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdm);
let n : integer = UInt(Rn);
let m : integer = UInt(Rdm);
let setflags : boolean = !InITBlock();
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	0	0	0	Rn				1	1	1	1	Rd				0	0	0	0	Rm			
op1												Ra								op2											

Encoding

MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}

MUL<c>.W <Rd>, <Rn>{, <Rm>} // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. If omitted, <Rd> is used.
<Rdm>	Is the second general-purpose source register holding the multiplier and the destination register, encoded in the "Rdm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : integer = SInt(R(n)); // operand1 = UInt(R[n]) produces the same final results
    let operand2 : integer = SInt(R(m)); // operand2 = UInt(R[m]) produces the same final results
    let result : integer = operand1 * operand2;
    R(d) = result[31:0];
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result[31:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN, MVNS (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register.
If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	1	1	1	1	1	S	(0)	(0)	(0)	(0)	Rd					imm12											
cond				opc							Rn																					

Encoding for the MVN variant

Applies when (S == 0)

MVN{<c>}{<q>}<Rd>, #<const>

Encoding for the MVNS variant

Applies when (S == 1)

MVNS{<c>}{<q>}<Rd>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	0	0	1	1	S	1	1	1	1	0	imm3				Rd				imm8							
op1							Rn																									

Encoding for the MVN variant

Applies when (S == 0)

MVN{<c>}{<q>}<Rd>, #<const>

Encoding for the MVNS variant

Applies when (S == 1)

MVNS{<c>}{<q>}<Rd>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

<Rd> For the "A1 MVN" and "A1 MVNS" variants: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the MVNS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the "T1 MVN" and "T1 MVNS" variants: is the general-purpose destination register, encoded in the "Rd" field.

<const> For the "A1 MVN" and "A1 MVNS" variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the "T1 MVN" and "T1 MVNS" variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm\_C(imm, PSTATE.C)
                                           else T32ExpandImm\_C(imm, PSTATE.C));

    let result : bits(32) = NOT(imm32);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUEXCEPTIONRETURN(result);
        else
            ALUWRITEPC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN, MVNS (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register.

If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	1	S	(0)	(0)	(0)	(0)	Rd				imm5				stype	0	Rm					
cond				opc				Rn																							

Encoding for the MVN, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

MVN{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MVN, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

Encoding for the MVNS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

MVNS{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MVNS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	1	Rm			Rd		
op															

Encoding

MVN<c>{<q>} <Rd>, <Rm> // (InITBlock())

MVNS{<q>} <Rd>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	0	1	0	0	1	1	S	1	1	1	1	(0)	imm3				Rd				imm2		stype		Rm						
op1												Rn																							

Encoding for the MVN, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

MVN{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MVN, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

MVN<c>.W <Rd>, <Rm> // (Inside IT block, and <Rd>, <Rm> can be represented in T1)

Encoding for the MVNS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

MVNS{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MVNS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

MVNS.W <Rd>, <Rm> // (Outside IT block, and <Rd>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	For the "A1 MVN, rotate right with extend", "A1 MVN, shift or rotate by value", "A1 MVNS, rotate right with extend", and "A1 MVNS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used: • For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. • For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. For the "T1", "T2 MVN, rotate right with extend", "T2 MVN, shift or rotate by value", "T2 MVNS, rotate right with extend", and "T2 MVNS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field.										
<Rm>	For the "A1 MVN, rotate right with extend", "A1 MVN, shift or rotate by value", "A1 MVNS, rotate right with extend", and "A1 MVNS, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T1", "T2 MVN, rotate right with extend", "T2 MVN, shift or rotate by value", "T2 MVNS, rotate right with extend", and "T2 MVNS, shift or rotate by value" variants: is the general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the source register, encoded in “stype”: <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the "A1 MVN, shift or rotate by value" and "A1 MVNS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.										

For the "T2 MVN, shift or rotate by value" and "T2 MVNS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift\_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;
```

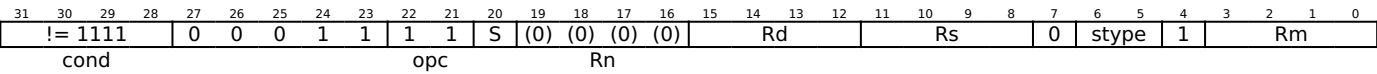
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN, MVNS (register-shifted register)

Bitwise NOT (register-shifted register) writes the bitwise inverse of a register-shifted register value to the destination register. It can optionally update the condition flags based on the result.

A1



Encoding for the Flag setting variant

Applies when (S == 1)

```
MVNS{<c>}{<q>}<Rd>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when (S == 0)

```
MVN{<c>}{<q>}<Rd>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.
- <shift> Is the type of shift to be applied to the second source register, encoded in "stype":

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let shift_n : integer = UInt(R(s)[7:0]);
  let (shifted, carry) : (bits(32), bit) = Shift_C[32](R(m), shift_t, shift_n, PSTATE.C);
  let result : bits(32) = NOT(shifted);
  R(d) = result;
  if setflags then
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit[32](result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
  end;
end;
```

NOP

No Operation does nothing. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a `NOP` instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, `NOP` instructions are not suitable for timing loops.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=	1	1	1	1	0	0	1	1	0	0	1	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	0
cond				R								imm4				imm12															

Encoding

`NOP{<c>} {<q>}`

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
hint															

Encoding

`NOP{<c>} {<q>}`

Decode for this encoding

```
// No additional decoding required
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	0
												hint				option															

Encoding

`NOP{<c>}.W`

Decode for this encoding

```
// No additional decoding required
```

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

ORN, ORNS (immediate)

Bitwise OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	0	0	1	1	S	!= 1111				0	imm3				Rd				imm8							
op1												Rn																				

Encoding for the Flag setting variant

Applies when (S == 1)

ORNS{<c>}{<q>}{<Rd>}, <Rn>, #<const>

Encoding for the Not flag setting variant

Applies when (S == 0)

ORN{<c>}{<q>}{<Rd>}, <Rn>, #<const>

Decode for all variants of this encoding

```
if Rn == '1111' then See("MVN (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
- <Rn> Is the general-purpose source register, encoded in the "Rn" field.
- <const> An immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm C(imm, PSTATE.C)
                                            else T32ExpandImm C(imm, PSTATE.C));

    let result : bits(32) = R(n) OR NOT(imm32);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

ORN, ORNS (register)

Bitwise OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	1	S	!	1	1	1	1	(0)	imm3													
op1												Rn																			

Encoding for the ORN, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

ORN{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ORN, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ORN{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the ORNS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

ORNS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ORNS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ORNS{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
if Rn == '1111' then See("MVN (register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift\_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) OR NOT(shifted);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR, ORRS (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 0 1 1 1				0 0 S				Rn				Rd				imm12											
cond								opc																							

Encoding for the ORR variant

Applies when (S == 0)

ORR{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the ORRS variant

Applies when (S == 1)

ORRS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	0	0	1	0	S	!= 1111				0	imm3				Rd				imm8							
op1												Rn																				

Encoding for the ORR variant

Applies when (S == 0)

ORR{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the ORRS variant

Applies when (S == 1)

ORRS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
if Rn == '1111' then See("MOV (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 ORR" and "A1 ORRS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
	For the "T1 ORR" and "T1 ORRS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For the "A1 ORR" and "A1 ORRS" variants: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
	For the "T1 ORR" and "T1 ORRS" variants: is the general-purpose source register, encoded in the "Rn" field.
<const>	For the "A1 ORR" and "A1 ORRS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values.
	For the "T1 ORR" and "T1 ORRS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm C(imm, PSTATE.C)
                                           else T32ExpandImm C(imm, PSTATE.C));

    let result : bits(32) = R(n) OR imm32;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR, ORRS (register)

Bitwise OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 0 0 1 1				0 0		S	Rn				Rd				imm5				stype		0	Rm					
cond										opc																					

Encoding for the ORR, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

```
ORR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORR, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

```
ORR{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Encoding for the ORRS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

```
ORRS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORRS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

```
ORRS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	0	Rm			Rdn		
op															

Encoding

```
ORR{<c>}{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

```
ORRS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	!= 1111				(0)	imm3			Rd			imm2			stype		Rm			
op1												Rn																			

Encoding for the ORR, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
ORR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORR, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ORR{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ORR<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Encoding for the ORRS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
ORRS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX
```

Encoding for the ORRS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ORRS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

```
ORRS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("Related encodings"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Data-processing \(shifted register\)](#)

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> For the "A1 ORR, rotate right with extend", "A1 ORR, shift or rotate by value", "A1 ORRS, rotate right with extend", and "A1 ORRS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ORRS variant, the instruction performs an exception return, that restores **PSTATE** from SPSR_<current_mode>.

For the "T2 ORR, rotate right with extend", "T2 ORR, shift or rotate by value", "T2 ORRS, rotate right with extend", and "T2 ORRS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For the "A1 ORR, rotate right with extend", "A1 ORR, shift or rotate by value", "A1 ORRS, rotate right with extend", and "A1 ORRS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For the "T2 ORR, rotate right with extend", "T2 ORR, shift or rotate by value", "T2 ORRS, rotate right with extend", and "T2 ORRS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For the "A1 ORR, rotate right with extend", "A1 ORR, shift or rotate by value", "A1 ORRS, rotate right with extend", and "A1 ORRS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T1", "T2 ORR, rotate right with extend", "T2 ORR, shift or rotate by value", "T2 ORRS, rotate right with extend", and "T2 ORRS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the "A1 ORR, shift or rotate by value" and "A1 ORRS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For the "T2 ORR, shift or rotate by value" and "T2 ORRS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

In T32 assembly:

- Outside an IT block, if ORRS <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORRS <Rd>, <Rn> had been written.
- Inside an IT block, if ORR<c> <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORR<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) OR shifted;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

ORR, ORRS (register-shifted register)

Bitwise OR (register-shifted register) performs a bitwise (inclusive) OR of a register value and a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	0	S	Rn				Rd				Rs				0	stype		1	Rm			
cond				opc																											

Encoding for the Flag setting variant

Applies when (S == 1)

ORRS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

ORR{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”:										
<table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>		stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<Rs>	Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C(32)(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) OR shifted;
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit(32)(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

PKHBT, PKHTB

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	0	0	Rn				Rd				imm5				tb	0	1	Rm				
cond																															

Encoding for the PKHBT variant

Applies when (tb == 0)

PKHBT{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, LSL #<imm>}

Encoding for the PKHTB variant

Applies when (tb == 1)

PKHTB{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, ASR #<imm>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let tbform : boolean = (tb == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(tb::'0', imm5);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	1	0	0	Rn				(0)	imm3				Rd				imm2	tb	0	Rm			
op1												S												T							

Encoding for the PKHBT variant

Applies when (tb == 0)

PKHBT{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, LSL #<imm>} // (tbform == FALSE)

Encoding for the PKHTB variant

Applies when (tb == 1)

PKHTB{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, ASR #<imm>} // (tbform == TRUE)

Decode for all variants of this encoding

```
if S == '1' || T == '1' then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let tbform : boolean = (tb == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(tb::'0', imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<imm>	For the "A1 PKHBT" and "A1 PKHTB" variants: the shift to apply to the value read from <Rm>, encoded in the "imm5" field. For PKHBT, it is one of:
omitted	No shift, encoded as 0b00000.
1-31	Left shift by specified number of bits, encoded as a binary number.
	For PKHTB, it is one of:
omitted	Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn> had been written.
1-32	Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.
Note	
An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.	
	For the "T1 PKHBT" and "T1 PKHTB" variants: the shift to apply to the value read from <Rm>, encoded in the "imm3:imm2" field. For PKHBT, it is one of:
omitted	No shift, encoded as 0b00000.
1-31	Left shift by specified number of bits, encoded as a binary number.
	For PKHTB, it is one of:
omitted	Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn> had been written.
1-32	Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.
Note	
An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.	

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = Shift{ } (R(m), shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
    R(d)[15:0] = if tbform then operand2[15:0] else R(n)[15:0];
    R(d)[31:16] = if tbform then R(n)[31:16] else operand2[31:16];
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

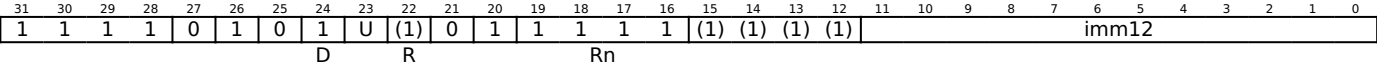
PLD (literal)

Preload Data (literal) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The effect of a PLD instruction is IMPLEMENTATION DEFINED. For more information, see [Preloading caches](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



Encoding

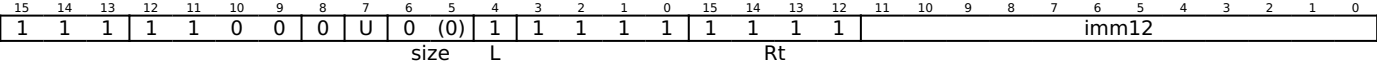
```
PLD{<c>}{<q>} <label> // (Normal form)

PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

T1



Encoding

```
PLD{<c>}{<q>} <label> // (Preferred syntax)

PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c>

For the "A1" variant: see [Standard assembler syntax fields](#). Must be AL or omitted.
For the "T1" variant: see [Standard assembler syntax fields](#).
- <q>

See [Standard assembler syntax fields](#).
- <label>

The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range −4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE.
- +/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	−
1	+
- <imm>

For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.
For the "T1" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                                else (AlignDownSize(PC32(),4) - imm32));
    Hint_PreloadData(address);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PLD, PLDW (immediate)

Preload Data (immediate) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

The effect of a `PLD` or `PLDW` instruction is IMPLEMENTATION DEFINED. For more information, see [Preloading caches](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	U	R	0	1	!= 1111				(1)	(1)	(1)	(1)	imm12											
D								Rn																							

Encoding for the Preload read variant

Applies when (R == 1)

```
PLD{<c>}{<q>} [<Rn> {, # {+/-}<imm>}]
```

Encoding for the Preload write variant

Applies when (R == 0)

```
PLDW{<c>}{<q>} [<Rn> {, # {+/-}<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
let is_pldw : boolean = (R == '0');
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	W	1	!= 1111				1	1	1	1	imm12											
L											Rn					Rt															

Encoding for the Preload read variant

Applies when (W == 0)

```
PLD{<c>}{<q>} [<Rn> {, # {+}<imm>}]
```

Encoding for the Preload write variant

Applies when (W == 1)

```
PLDW{<c>}{<q>} [<Rn> {, # {+}<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = TRUE;
let is_pldw : boolean = (W == '1');
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	0	0	0	0	0	W	1	!= 1111				1	1	1	1	1	1	0	0	imm8									
L											Rn					Rt					P		U										

Encoding for the Preload read variant

Applies when (W == 0)

```
PLD{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Encoding for the Preload write variant

Applies when (W == 1)

PLDW{<c>}{<q>} [<Rn> {, #-<imm>}]

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let add : boolean = FALSE;
let is_pldw : boolean = (W == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	For the "A1 Preload read" and "A1 Preload write" variants: see Standard assembler syntax fields . Must be AL or omitted. For the "T1 Preload read", "T1 Preload write", "T2 Preload read", and "T2 Preload write" variants: see Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see PLD (literal) .						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”: <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Preload read" and "A1 Preload write" variants: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T1 Preload read" and "T1 Preload write" variants: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T2 Preload read" and "T2 Preload write" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.						
+	Specifies the offset is added to the base register.						

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let address : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
  if is_pldw then
    Hint_PreloadDataForWrite(address);
  else
    Hint_PreloadData(address);
  end;
end;
```

PLD, PLDW (register)

Preload Data (register) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

The effect of a `PLD` or `PLDW` instruction is IMPLEMENTATION DEFINED. For more information, see [Preloading caches](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	U	R	0	1	Rn				(1)	(1)	(1)	(1)	imm5					stype	0	Rm				
D																															

Encoding for the Preload read, optional shift or rotate variant

Applies when $(R == 1 \ \&\& \ !(\text{imm5} == 00000 \ \&\& \ \text{stype} == 11))$

`PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]`

Encoding for the Preload read, rotate right with extend variant

Applies when $(R == 1 \ \&\& \ \text{imm5} == 00000 \ \&\& \ \text{stype} == 11)$

`PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]`

Encoding for the Preload write, optional shift or rotate variant

Applies when $(R == 0 \ \&\& \ !(\text{imm5} == 00000 \ \&\& \ \text{stype} == 11))$

`PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]`

Encoding for the Preload write, rotate right with extend variant

Applies when $(R == 0 \ \&\& \ \text{imm5} == 00000 \ \&\& \ \text{stype} == 11)$

`PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]`

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = (U == '1');
let is_pldw : boolean = (R == '0');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
if m == 15 || (n == 15 && is_pldw) then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	W	1	!= 1111				1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
L											Rn					Rt					imm2					Rm					

Encoding for the Preload read variant

Applies when $(W == 0)$

`PLD{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]`

Encoding for the Preload write variant

Applies when $(W == 1)$

`PLDW{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]`

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = TRUE;
let is_pldw : boolean = (W == '1');
let shift_t : SRTType = SRTType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	For the "Preload read, optional shift or rotate", "Preload read, rotate right with extend", "Preload write, optional shift or rotate", and "Preload write, rotate right with extend" variants: see Standard assembler syntax fields . <c> must be AL or omitted.										
	For the "Preload read" and "Preload write" variants: see Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rn>	For the "Preload read, optional shift or rotate", "Preload read, rotate right with extend", "Preload write, optional shift or rotate", and "Preload write, rotate right with extend" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used.										
	For the "Preload read" and "Preload write" variants: is the general-purpose base register, encoded in the "Rn" field.										
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+				
U	+/-										
0	-										
1	+										
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the index register, encoded in "stype": <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the "Preload read, optional shift or rotate" and "Preload write, optional shift or rotate" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.										
	For the "Preload read" and "Preload write" variants: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.										
+	Specifies the index register is added to the base register.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{(R(m), shift_t, shift_n, PSTATE.C);
    let address : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
    end;
end;
```

PLI (immediate, literal)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	U	1	0	1	Rn				(1)	(1)	(1)	(1)	imm12											
D								R																							

Encoding

```
PLI{<c>}{<q>} [<Rn> {, #{+/-}<imm>}] // (Preferred syntax)

PLI{<c>}{<q>} <label> // (Normal form)

PLI{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	0	1	!= 1111			1	1	1	1	imm12												
size												Rn				Rt															

Encoding

```
PLI{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("encoding T3"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = TRUE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
1	1	1	1	1	0	0	1	0	0	0	1	!= 1111				1	1	1	1	1	1	0	0	imm8													
size												Rn				Rt				P		U		W													

Encoding

```
PLI{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("encoding T3"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let add : boolean = FALSE;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	1	1	1	1	1	imm12															
size																Rt																				

Encoding

```
PLI{<c>}{<q>} <label> // (Preferred syntax)

PLI{<c>}{<q>} [PC, #<+/->{<imm>}] // (Alternative syntax)
```

Decode for this encoding

```
let n : integer = 15;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	For the "A1" variant: see Standard assembler syntax fields . Must be AL or omitted. For the "T1", "T2", and "T3" variants: see Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table> <tr> <th>U</th><th>+/-</th></tr> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1" variant: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T1" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T2" variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field. For the "T3" variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.						
<label>	The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range –4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE.						
+	Specifies the offset is added to the base register.						

For the literal forms of the instruction, encoding T3 is used, or Rn is encoded as 0b1111 in encoding A1, to indicate that the PC is the base register. The alternative literal syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = if n == 15 then AlignDownSize(PC32(),4) else R(n);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
end;
```

PLI (register)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see [Preloading caches](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	0	U	1	0	1	Rn				(1)	(1)	(1)	(1)	imm5				stype		0	Rm				
D								o2																							

Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]

Encoding for the Shift or rotate by value variant

Applies when (!(imm5 == 00000 && stype == 11))

PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = (U == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
if m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	!= 1111				1	1	1	1	0	0	0	0	0	0	0	imm2	Rm			
size							Rn				Rt																				

Encoding

PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]

Decode for this encoding

```
if Rn == '1111' then See("PLI (immediate, literal)"); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = TRUE;
let shift_t : SRTType = SRTType_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c>

For the "Rotate right with extend" and "Shift or rotate by value" variants: see [Standard assembler syntax fields](#). <c> must be AL or omitted.
- <q>

See [Standard assembler syntax fields](#).
- <Rn>

Is the general-purpose base register, encoded in the "Rn" field.
- +/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+
- <Rm>

Is the general-purpose index register, encoded in the "Rm" field.

<shift>

Is the type of shift to be applied to the index register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the "Shift or rotate by value" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For the "T1" variant: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

+

Specifies the index register is added to the base register.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let offset : bits(32) = Shift{(R(m), shift_t, shift_n, PSTATE.C);
  let address : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
  Hint_PreloadInstr(address);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

POP

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	P	register list							

L

Encoding

```
POP{<c>}{<q>} <registers> // (Preferred syntax)

LDM{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

Decode for this encoding

```
let registers : bits(16) = P::'0000000'::register_list;
let UnalignedAllowed : boolean = FALSE;
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
if registers[15] == '1' && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<registers>	Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0. If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(13);
    for i = 0 to 14 do
        if registers[i] == '1' then
            R(i) = if UnalignedAllowed then MemU{32}(address) else MemA{32}(address);
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        if UnalignedAllowed then
            if address[1:0] == '00' then
                LoadWritePC(MemU{32}(address));
            else
                UnpredictableProcedure();
            end;
        else
            LoadWritePC(MemA{32}(address));
        end;
    end;
    if registers[13] == '0' then R(13) = R(13) + 4*BitCount(registers); end;
    if registers[13] == '1' then R(13) = ARBITRARY : bits(32); end;
end;
```


POP (multiple registers)

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

This is an alias of [LDM, LDMIA, LDMFD](#). This means:

- The encodings in this description are named to match the encodings of [LDM, LDMIA, LDMFD](#).
- The description of [LDM, LDMIA, LDMFD](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	0	0	0	1	0	1	1	1	1	0	1	register_list															
cond				P		U		op		W		L		Rn																	

Encoding for the A1 variant

POP{<c>} {<q>} <registers>

is equivalent to

LDM{IA} {<c>} {<q>} SP!, <registers>

and is the preferred disassembly when `BitCount(register_list) > 1`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	1	1	1	1	0	1	P	M	register_list													
opc								W		L		Rn																			

Encoding for the T2 variant

POP{<c>} {<q>} <registers>

POP{<c>}.W <registers> // (M == 0 && register_list[13:8] == '000000')

is equivalent to

LDM{IA} {<c>}.W SP!, <registers>

and is the preferred disassembly when `BitCount(P :: M :: register_list) > 1`.

Assembler Symbols

- <c>

See [Standard assembler syntax fields](#).
- <q>

See [Standard assembler syntax fields](#).
- <registers>

For the "A1" variant: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#). If the SP is in the list, the value of the SP after such an instruction is UNKNOWN. The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#). Arm deprecates the use of this instruction with both the LR and the PC in the list.

For the "T2" variant: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#). The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#). If the PC is in the list:

 - The LR must not be in the list.
 - The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

The description of [LDM, LDMIA, LDMFD](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [LDM, LDMIA, LDMFD](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

POP (single register)

Pop Single Register from Stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point just above the loaded data.

This is an alias of [LDR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [LDR \(immediate\)](#).
- The description of [LDR \(immediate\)](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T4](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=	1	1	1	1	0	1	0	0	1	0	0	1	1	1	0	1															
cond				P		U	op2	W	op1	Rn						imm12															

Encoding for the Post-indexed variant

POP{<c>} {<q>} <single_register_list>

is equivalent to

[LDR](#){<c>} {<q>} <Rt>, [SP], #4

and is always the preferred disassembly.

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	0	1	0	1	1	1	0	1															
size				L		Rn						P				U	W	imm8													

Encoding for the Post-indexed variant

POP{<c>} {<q>} <single_register_list>

is equivalent to

[LDR](#){<c>} {<q>} <Rt>, [SP], #4

and is always the preferred disassembly.

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <single_register_list> Is the general-purpose register <Rt> to be loaded surrounded by { and }.

Operation

The description of [LDR \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [LDR \(immediate\)](#) gives the operational information for this instruction.

PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions. For more information and details of the semantics, see *Physical Speculative Store Bypass Barrier (PSSBB)*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	0	1	0	0				
																								opcode							option				

Encoding

PSSBB{<q>}

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	0	1	0	0
																opc				option											

Encoding

PSSBB{<q>}

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
```

Assembler Symbols

<q> See *Standard assembler syntax fields*.

Operation

```
EncodingSpecificOperations();
SpeculativeStoreBypassBarrierToPA();
```

PUSH

Push Multiple Registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	M	register_list							
								L							

Encoding

```
PUSH{<c>}{<q> <registers> // (Preferred syntax)

STMDB{<c>}{<q> SP!, <registers> // (Alternate syntax)
```

Decode for this encoding

```
let registers : bits(16) = '0'::M::'000000'::register_list;
let UnalignedAllowed : boolean = FALSE;
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<registers>	Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(13) - 4*BitCount(registers);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == 13 && i != LowestSetBit(registers) then // Only possible for encoding A1
                MemA{32}(address) = ARBITRARY : bits(32);
            else
                if UnalignedAllowed then
                    MemU{32}(address) = R(i);
                else
                    MemA{32}(address) = R(i);
                end;
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then // Only possible for encoding A1 or A2
        if UnalignedAllowed then
            MemU{32}(address) = PCStoreValue();
        else
            MemA{32}(address) = PCStoreValue();
        end;
    end;
    R(13) = R(13) - 4*BitCount(registers);
end;
```

PUSH (multiple registers)

Push multiple registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

This is an alias of [STMDB, STMFD](#). This means:

- The encodings in this description are named to match the encodings of [STMDB, STMFD](#).
- The description of [STMDB, STMFD](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	0	0	1	0	0	1	0	1	1	0	1	register_list															
cond				P		U		op		W		L		Rn																	

Encoding for the A1 variant

PUSH{<c>} {<q>} <registers>

is equivalent to

[STMDB](#){<c>} {<q>} SP!, <registers>

and is the preferred disassembly when `BitCount(register_list) > 1`.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	1	0	1	1	0	1	(0)	M	register_list													
opc							W		L		Rn				P																

Encoding for the T1 variant

PUSH{<c>} {<q>} <registers>

PUSH{<c>}.W <registers> // (register_list[13:8] == '000000')

is equivalent to

[STMDB](#){<c>} {<q>} SP!, <registers>

and is the preferred disassembly when `BitCount(M :: register_list) > 1`.

Assembler Symbols

- <c>
- See [Standard assembler syntax fields](#).
- <q>
- See [Standard assembler syntax fields](#).
- <registers>
- For the "A1" variant: is a list of two or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#). The SP and PC can be in the list. However:

- Arm deprecates the use of instructions that include the PC in the list.
- If the SP is in the list, and it is not the lowest-numbered register in the list, the instruction stores an UNKNOWN value for the SP.

For the "T1" variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#). The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation

The description of [STMDB, STMFD](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [STMDB, STMFD](#) gives the operational information for this instruction.

PUSH (single register)

Push Single Register to Stack stores a single general-purpose register to the stack, storing to the 32-bit word below the address in SP, and updates SP to point to the start of the stored data.

This is an alias of [STR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [STR \(immediate\)](#).
- The description of [STR \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T4](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111				0	1	0	1		0	1	0	1	1	0	1	Rt				0				0	0	0	0	0	0	0	1	0	0
cond				P		U		op2		W		op1		Rn				imm12															

Encoding for the Pre-indexed variant

`PUSH{<c>} {<q>} <single_register_list>`
is equivalent to
`STR{<c>} {<q>} <Rt>, [SP, #-4] !`
and is always the preferred disassembly.

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	1	1	1	0	0	0	0	1	0	0	1	1	0	1					1	1	0	1	0	0	0	0	0	1	0	0								
size												L				Rn				P				U				W				imm8							

Encoding for the Pre-indexed variant

`PUSH{<c>} {<q>} <single_register_list> // (Standard syntax)`
is equivalent to
`STR{<c>} {<q>} <Rt>, [SP, #-4] !`
and is always the preferred disassembly.

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <single_register_list> Is the general-purpose register <Rt> to be stored surrounded by { and }.

Operation

The description of [STR \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [STR \(immediate\)](#) gives the operational information for this instruction.

QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range -2^{31} to $(2^{31} - 1)$, and writes the result to the destination register. If saturation occurs, it sets *PSTATE.Q* to 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0	0	0	Rn				Rd				(0) (0) (0) (0)				0	1	0	1	Rm			
cond										opc																					

Encoding

QADD{<c>}{<q> } {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	0	Rn				1	1	1	1	Rd				1	0	0	0	Rm			
op1															op2																

Encoding

QADD{<c>}{<q> } {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the first general-purpose source register, encoded in the "Rm" field.
- <Rn> Is the second general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (r, sat) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) + SInt(R(n)));
    R(d) = r;
    if sat then
        PSTATE.Q = '1';
    end;
end;
```

QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	1	0	Rn			Rd			(1)	(1)	(1)	(1)	0	0	0	1	Rm					
cond				op1								B				op2															

Encoding

QADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rn				1	1	1	1	Rd				0	0	0	1	Rm			
op1												U												H				S			

Encoding

QADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[15:0]) + SInt(R(m)[15:0]);
    let sum2 : integer = SInt(R(n)[31:16]) + SInt(R(m)[31:16]);
    R(d)[15:0] = SignedSat{16}(sum1);
    R(d)[31:16] = SignedSat{16}(sum2);
end;
```

QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	1	0	Rn				Rd				(1)	(1)	(1)	(1)	1	0	0	1	Rm			
cond				op1																B				op2							

Encoding

QADD8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	0	1	0	0	0	Rn				1	1	1	1	Rd				0	0	0	1	Rm							
								op1																U				H				S			

Encoding

QADD8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

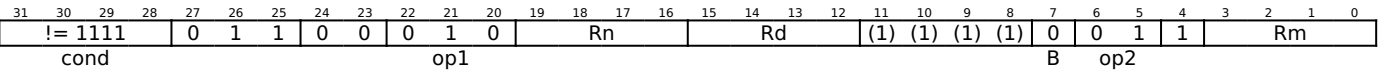
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[7:0]) + SInt(R(m)[7:0]);
    let sum2 : integer = SInt(R(n)[15:8]) + SInt(R(m)[15:8]);
    let sum3 : integer = SInt(R(n)[23:16]) + SInt(R(m)[23:16]);
    let sum4 : integer = SInt(R(n)[31:24]) + SInt(R(m)[31:24]);
    R(d)[7:0] = SignedSat{8}(sum1);
    R(d)[15:8] = SignedSat{8}(sum2);
    R(d)[23:16] = SignedSat{8}(sum3);
    R(d)[31:24] = SignedSat{8}(sum4);
end;
```

QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



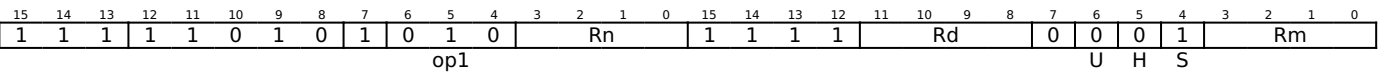
Encoding

QASX{<c>}{<q>}{<Rd>, <Rn>, <Rm>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

QASX{<c>}{<q>}{<Rd>, <Rn>, <Rm>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = SInt(R(n)[15:0]) - SInt(R(m)[31:16]);
    let sum : integer = SInt(R(n)[31:16]) + SInt(R(m)[15:0]);
    R(d)[15:0] = SignedSat{16}(diff);
    R(d)[31:16] = SignedSat{16}(sum);
end;
```

QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation occurs in either operation, it sets *PSTATE.Q* to 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	0	0	Rn				Rd				(0) (0) (0) (0)				0	1	0	1	Rm			
cond								opc																							

Encoding

QDADD{<c>}{<q>} {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	0	Rn				1	1	1	1	Rd				1	0	0	1	Rm			
op1																op2															

Encoding

QDADD{<c>}{<q>} {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the first general-purpose source register, encoded in the "Rm" field.
- <Rn> Is the second general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (doubled, sat1) : (bits(32), boolean) = SignedSatQ{32}(2 * SInt(R(n)));
    let (r, sat2) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) + SInt(doubled));
    R(d) = r;
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```

QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation occurs in either operation, it sets *PSTATE.Q* to 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0	0	0	1	0	1	1	0	Rn				Rd				(0)	(0)	(0)	(0)	0	1	0	1	Rm							
cond												opc																							

Encoding

QDSUB{<c>}{<q>} {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	0	Rn				1	1	1	1	Rd				1	0	1	1	Rm			
op1																op2															

Encoding

QDSUB{<c>}{<q>} {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the first general-purpose source register, encoded in the "Rm" field.
- <Rn> Is the second general-purpose source register, encoded in the "Rn" field.

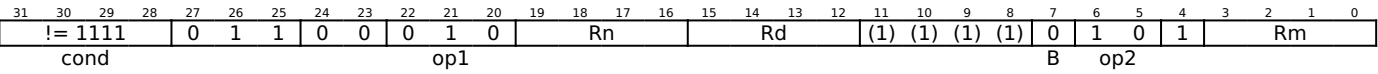
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (doubled, sat1) : (bits(32), boolean) = SignedSatQ{32}(2 * SInt(R(n)));
    let (r, sat2) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) - SInt(doubled));
    R(d) = r;
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```

QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



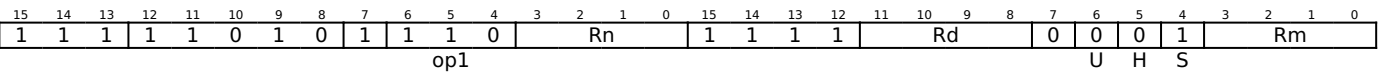
Encoding

QSAX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

QSAX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = SInt(R(n)[15:0]) + SInt(R(m)[31:16]);
    let diff : integer = SInt(R(n)[31:16]) - SInt(R(m)[15:0]);
    R(d)[15:0] = SignedSat{16}(sum);
    R(d)[31:16] = SignedSat{16}(diff);
end;
```

QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$, and writes the result to the destination register. If saturation occurs, it sets *PSTATE.Q* to 1. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0	1	0	Rn				Rd				(0)	(0)	(0)	(0)	0	1	0	1	Rm			
cond										opc																					

Encoding

QSUB{<c>}{<q> } {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	0	Rn				1	1	1	1	Rd				1	0	1	0	Rm			
op1												op2																			

Encoding

QSUB{<c>}{<q> } {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the first general-purpose source register, encoded in the "Rm" field.
- <Rn> Is the second general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (r, sat) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) - SInt(R(n)));
    R(d) = r;
    if sat then
        PSTATE.Q = '1';
    end;
end;
```


QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	1	0	Rn			Rd			(1) (1) (1) (1)			0	1	1	1	Rm						
cond				op1								B				op2															

Encoding

QSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	1	Rn			1	1	1	1	Rd			0	0	0	1	Rm					
op1												U												H			S				

Encoding

QSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[15:0]) - SInt(R(m)[15:0]);
    let diff2 : integer = SInt(R(n)[31:16]) - SInt(R(m)[31:16]);
    R(d)[15:0] = SignedSat{16}(diff1);
    R(d)[31:16] = SignedSat{16}(diff2);
end;
```

QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	1	0	Rn				Rd				(1)	(1)	(1)	(1)	1	1	1	1	Rm			
cond				op1																B				op2							

Encoding

QSUB8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	0	1	1	0	0	Rn				1	1	1	1	Rd				0	0	0	1	Rm							
								op1																U				H				S			

Encoding

QSUB8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[7:0]) - SInt(R(m)[7:0]);
    let diff2 : integer = SInt(R(n)[15:8]) - SInt(R(m)[15:8]);
    let diff3 : integer = SInt(R(n)[23:16]) - SInt(R(m)[23:16]);
    let diff4 : integer = SInt(R(n)[31:24]) - SInt(R(m)[31:24]);
    R(d)[7:0] = SignedSat{8}(diff1);
    R(d)[15:8] = SignedSat{8}(diff2);
    R(d)[23:16] = SignedSat{8}(diff3);
    R(d)[31:24] = SignedSat{8}(diff4);
end;
```

RBIT

Reverse Bits reverses the bit order in a 32-bit register.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	1	(1)	(1)	(1)	(1)	Rd				(1) (1) (1) (1)				0	0	1	1	Rm				
cond								o1								o2															

Encoding

RBIT{<c>}{<q> <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rn				1	1	1	1	Rd				1	0	1	0	Rm			
								op1																op2							

Encoding

RBIT{<c>}{<q> <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `m != n`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction executes with the additional decode: `m = UInt(Rn)`.
 - The instruction executes with the additional decode: `m = UInt(Rm)`.
 - The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1" variant: is the general-purpose source register, encoded in the "Rm" field. For the "T1" variant: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    for i = 0 to 31 do
        result[31-i] = R(m)[i];
    end;
    R(d) = result;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

REV

Byte-Reverse Word reverses the byte order in a 32-bit register.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	1	1	(1)	(1)	(1)	(1)	Rd				(1) (1) (1) (1)				0	0	1	1	Rm			
cond								o1								o2															

Encoding

REV{<c>} {<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	0	Rm			Rd		
op															

Encoding

REV{<c>} {<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rn				1	1	1	1	Rd				1	0	0	0	Rm			
op1												op2																			

Encoding

REV{<c>} {<q>} <Rd>, <Rm>

REV{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `m != n`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction executes with the additional decode: `m = UInt(Rn)`;
 - The instruction executes with the additional decode: `m = UInt(Rm)`;
 - The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1" and "T1" variants: is the general-purpose source register, encoded in the "Rm" field.

For the "T2" variant: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    result[31:24] = R(m)[7:0];
    result[23:16] = R(m)[15:8];
    result[15:8]  = R(m)[23:16];
    result[7:0]   = R(m)[31:24];
    R(d) = result;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV16

Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
!= 1111				0	1	1	0	1	0	1	1	(1)	(1)	(1)	(1)	Rd				(1)				(1)	(1)	(1)	1	0	1	1	Rm			
cond								o1								o2																		

Encoding

REV16{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	1	Rm			Rd		
op															

Encoding

REV16{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rn				1	1	1	1	Rd				1	0	0	1	Rm			
op1												op2																			

Encoding

REV16{<c>}{<q>} <Rd>, <Rm>

REV16{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `m != n`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction executes with the additional decode: `m = UInt(Rn);`.
 - The instruction executes with the additional decode: `m = UInt(Rm);`.
 - The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1" and "T1" variants: is the general-purpose source register, encoded in the "Rm" field.

For the "T2" variant: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    result[31:24] = R(m)[23:16];
    result[23:16] = R(m)[31:24];
    result[15:8]  = R(m)[7:0];
    result[7:0]  = R(m)[15:8];
    R(d) = result;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign-extends the result to 32 bits. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	1	(1)	(1)	(1)	(1)	Rd				(1) (1) (1) (1)				1	0	1	1	Rm				
cond								o1								o2															

Encoding

REVSH{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	1	1	Rm			Rd		
op															

Encoding

REVSH{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rn				1	1	1	1	Rd				1	0	1	1	Rm			
op1												op2																			

Encoding

REVSH{<c>}{<q>} <Rd>, <Rm>

REVSH{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `m != n`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction executes with the additional decode: `m = UInt(Rn)`;
 - The instruction executes with the additional decode: `m = UInt(Rm)`;
 - The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1" and "T1" variants: is the general-purpose source register, encoded in the "Rm" field.

For the "T2" variant: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    result[31:8] = SignExtend{24} (R(m)[7:0]);
    result[7:0]  = R(m)[15:8];
    R(d) = result;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RFE, RFEDA, RFEDB, RFEIA, RFEIB

Return From Exception loads two consecutive memory locations using an address in a base register:

- The word loaded from the lower address is treated as an instruction address. The PE branches to it.
- The word loaded from the higher address is used to restore *PSTATE*. This word must be in the format of an SPSR.

An address adjusted by the size of the data loaded can optionally be written back to the base register.

The PE checks the value of the word loaded from the higher address for an illegal return event. See *Illegal return events from AArch32 state*.

RFE is UNDEFINED in Hyp mode and CONSTRAINED UNPREDICTABLE in User mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	0	W	1	Rn				(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
S										L		op										mode									

Encoding for the Decrement After variant

Applies when (P == 0 && U == 0)

```
RFEDA{<c>}{<q>} <Rn>{!} // (Preferred syntax)
RFEFA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Full Ascending stack)
```

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0)

```
RFEDB{<c>}{<q>} <Rn>{!} // (Preferred syntax)
RFEEA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Empty Ascending stack)
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
RFE{IA}{<c>}{<q>} <Rn>{!} // (Preferred syntax)
RFEFD{<c>}{<q>} <Rn>{!} // (Alternate syntax, Full Descending stack)
```

Encoding for the Increment Before variant

Applies when (P == 1 && U == 1)

```
RFEIB{<c>}{<q>} <Rn>{!} // (Preferred syntax)
RFEED{<c>}{<q>} <Rn>{!} // (Alternate syntax, Empty Descending stack)
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let wback : boolean = (W == '1');
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	0	W	1	Rn				(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
opc							L		P		M	register list																			

Encoding

```
RFEDB{<c>}{<q>} <Rn>{!} // (Outside or last in IT block, preferred syntax)
RFEFA{<c>}{<q>} <Rn>{!} // (Outside or last in IT block, alternate syntax, Full Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let wback : boolean = (W == '1');
let increment : boolean = FALSE;
let wordhigher : boolean = FALSE;
if n == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	0	W	1					(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
opc											L	Rn				P	M	register_list													

Encoding

```
RFE{IA}{<c>}{<q>}<Rn>{!} // (Outside or last in IT block, preferred syntax)
RFEFD{<c>}{<q>}<Rn>{!} // (Outside or last in IT block, alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let wback : boolean = (W == '1');
let increment : boolean = TRUE;
let wordhigher : boolean = FALSE;
if n == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	For the "A1 Decrement After", "A1 Decrement Before", "A1 Increment After", and "A1 Increment Before" variants: see Standard assembler syntax fields . <c> must be AL or omitted. For the "T1" and "T2" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
IA	For the "Increment After" variant: is an optional suffix to indicate the Increment After variant. For the "T2" variant: is an optional suffix for the Increment After form.

RFEFA, RFEEA, RFEFD, and RFEED are pseudo-instructions for RFEDA, RFEDB, RFEIA, and RFEIB respectively, referring to their use for popping data from Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        Undefined();
    elsif PSTATE.EL == EL0 then
        UnpredictableProcedure(); // UNDEFINED or NOP
    else
        var address : bits(32) = if increment then R(n) else R(n)-8;
        if wordhigher then address = address+4; end;
        let new_pc_value : bits(32) = MemA{32}(address);
        let spsr : bits(32) = MemA{32}(address+4);
        if wback then R(n) = if increment then R(n)+8 else R(n)-8; end;
        AArch32_ExceptionReturn(new_pc_value, spsr);
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ROR (immediate)

Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd					!= 0000				1	1	0	Rm			
cond				opc				S				Rn				imm5					stype										

Encoding for the MOV, shift or rotate by value variant

ROR{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	0	1	0	1	0	0	1	0	0	1	1	1	1	(0)	imm3				Rd				imm2		1	1	Rm					
op1							S					Rn																	stype					

Encoding for the MOV, shift or rotate by value variant

Applies when (! (imm3 == 000 && imm2 == 00))

ROR{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Rd>	For the "A1 MOV, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <i>Pseudocode description of operations on the AArch32 general-purpose registers and the PC</i> . For the "T3 MOV, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1 MOV, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T3 MOV, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For the "A1 MOV, shift or rotate by value" variant: is the shift amount, in the range 1 to 31, encoded in the "imm5" field. For the "T3 MOV, shift or rotate by value" variant: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd				Rs				0	1	1	1	Rm			
cond				opc				S				Rn												stype							

Encoding for the Not flag setting variant

ROR{<c>} {<q>} {<Rd>, }<Rm>, <Rs>

is equivalent to

MOV{<c>} {<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	1	Rs			Rdm		
op															

Encoding for the Rotate right variant

ROR<c>{<q>} {<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs>

and is the preferred disassembly when `InitBlock()`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	0	0	1	1	0	Rm				1	1	1	1	Rd				0	0	0	0	Rs							
stype												S																							

Encoding for the Not flag setting variant

ROR{<c>} {<q>} {<Rd>, }<Rm>, <Rs>

ROR<c>.W {<Rd>, }<Rm>, <Rs> // (InitBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))

is equivalent to

MOV{<c>} {<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RORS (immediate)

Rotate Right, setting flags (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 0		0		1 1		0 1		1		(0) (0) (0) (0)		Rd				!= 00000				1 1		0		Rm			
cond								opc				S		Rn								imm5				stype					

Encoding for the MOVS, shift or rotate by value variant

RORS{<c>}{<q>}{<Rd>}, <Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	(0)	imm3				Rd				imm2				1	1	Rm					
op1							S	Rn																											
																stype																			

Encoding for the MOVS, shift or rotate by value variant

Applies when (! (imm3 == 000 && imm2 == 00))

RORS{<c>}{<q>}{<Rd>}, <Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Rd>	For the "A1 MOVS, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores <i>PSTATE</i> from SPSR_<current_mode>. For the "T3 MOVS, shift or rotate by value" variant: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1 MOVS, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T3 MOVS, shift or rotate by value" variant: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For the "A1 MOVS, shift or rotate by value" variant: is the shift amount, in the range 1 to 31, encoded in the "imm5" field. For the "T3 MOVS, shift or rotate by value" variant: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RORS (register)

Rotate Right, setting flags (register) provides the value of the contents of a register rotated by a variable number of bits, and updates the condition flags based on the result. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd				Rs				0	1	1	1	Rm			
cond								opc				S				Rn								stype							

Encoding for the Flag setting variant

RORS{<c>}{<q>}{<Rd>}, <Rm>, <Rs>

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	1	Rs			Rdm		
op															

Encoding for the Rotate right variant

RORS{<q>}{<Rdm>}, <Rdm>, <Rs>

is equivalent to

MOVS{<q>}<Rdm>, <Rdm>, ROR <Rs>

and is the preferred disassembly when `!InITBlock()`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	1	1	Rm				1	1	1	1	Rd				0	0	0	0	Rs			
								stype								S															

Encoding for the Flag setting variant

RORS{<c>}{<q>}{<Rd>}, <Rm>, <Rs>

RORS.W {<Rd>}, <Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RRX

Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T3](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd				0	0	0	0	0	1	1	0	Rm			
cond				opc				S				Rn				imm5				stypc											

Encoding for the MOV, rotate right with extend variant

RRX{<c>}{<q>}{<Rd>, }<Rm>

is equivalent to

MOV{<c>}{<q>}<Rd>, <Rm>, RRX

and is always the preferred disassembly.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	0	1	1	1	1	(0)	0	0	0	Rd				0	0	1	1	Rm			
op1				S				Rn				imm3								imm2				stypc							

Encoding for the MOV, rotate right with extend variant

RRX{<c>}{<q>}{<Rd>, }<Rm>

is equivalent to

MOV{<c>}{<q>}<Rd>, <Rm>, RRX

and is always the preferred disassembly.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 MOV, rotate right with extend" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC . For the "T3 MOV, rotate right with extend" variant: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For the "A1 MOV, rotate right with extend" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T3 MOV, rotate right with extend" variant: is the general-purpose source register, encoded in the "Rm" field.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

RRXS

Rotate Right with Extend, setting flags provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].
If the destination register is not the PC, this instruction updates the condition flags based on the result, and bit[0] is shifted into the Carry flag. The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T3](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111				0	0	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd				0				0	0	0	0	1	1	0	Rm		
cond								opc				S	Rn									imm5					stype						

Encoding for the MOVS, rotate right with extend variant

RRXS{<c>}{<q>}{<Rd>, }<Rm>

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, RRX

and is always the preferred disassembly.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	1	(0)	0	0	0	Rd					0	0	1	1	Rm		
op1				S				Rn				imm3					imm2					stype									

Encoding for the MOVS, rotate right with extend variant

RRXS{<c>}{<q>}{<Rd>, }<Rm>

is equivalent to

MOVS{<c>}{<q>}<Rd>, <Rm>, RRX

and is always the preferred disassembly.

Assembler Symbols

- <c>

See *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <Rd>

For the "A1 MOVS, rotate right with extend" variant: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.
For the "T3 MOVS, rotate right with extend" variant: is the general-purpose destination register, encoded in the "Rd" field.
- <Rm>

For the "A1 MOVS, rotate right with extend" variant: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For the "T3 MOVS, rotate right with extend" variant: is the general-purpose source register, encoded in the "Rm" field.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [MOV, MOVS \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSB, RSBS (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	0	1	1	S	Rn				Rd				imm12											
cond				opc																											

Encoding for the RSB variant

Applies when (S == 0)

RSB{<c>}{<q>}{<Rd>, }<Rn>, #<const>

Encoding for the RSBS variant

Applies when (S == 1)

RSBS{<c>}{<q>}{<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	1	Rn			Rd		
op															

Encoding

RSB<c>{<q>}{<Rd>, }<Rn>, #0 // (InITBlock())

RSBS{<q>}{<Rd>, }<Rn>, #0 // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = Zeros{}; // immediate = #0
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	1	1	1	0	S	Rn				0	imm3				Rd				imm8							
op1																																

Encoding for the RSB variant

Applies when (S == 0)

RSB{<c>}{<q>}{<Rd>, }<Rn>, #<const>

RSB<c>.W {<Rd>, }<Rn>, #0 // (Inside IT block)

Encoding for the RSBS variant

Applies when (S == 1)


```
RSBS{<c>}{<q>}{<Rd>,}<Rn>, #<const>

RSBS.W {<Rd>,}<Rn>, #0 // (Outside IT block)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 RSB" and "A1 RSBS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the RSBS variant, the instruction performs an exception return, that restores <i>PSTATE</i> from <i>SPSR_<current_mode></i>. For the "T1", "T2 RSB", and "T2 RSBS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For the "A1 RSB" and "A1 RSBS" variants: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T1", "T2 RSB", and "T2 RSBS" variants: is the general-purpose source register, encoded in the "Rn" field.
<const>	For the "A1 RSB" and "A1 RSBS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values. For the "T2 RSB" and "T2 RSBS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), imm32, '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSB, RSBS (register)

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register. If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result. The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	1	1	S	Rn				Rd				imm5				stype		0	Rm				
cond								opc																							

Encoding for the RSB, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

RSB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the RSB, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

RSB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the RSBS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

RSBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the RSBS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

RSBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	0	1	1	1	1	0	S	Rn				(0)	imm3				Rd				imm2		stype		Rm			
op1																																

Encoding for the RSB, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

RSB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the RSB, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

RSB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the RSBS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

RSBS{<c>}{<q>}{<Rd>, <Rn>, <Rm>, RRX

Encoding for the RSBS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

RSBS{<c>}{<q>}{<Rd>, <Rn>, <Rm>{, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	For the "A1 RSB, rotate right with extend", "A1 RSB, shift or rotate by value", "A1 RSBS, rotate right with extend", and "A1 RSBS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.										
<Rn>	For the "T1 RSB, rotate right with extend", "T1 RSB, shift or rotate by value", "T1 RSBS, rotate right with extend", and "T1 RSBS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. For the "A1 RSB, rotate right with extend", "A1 RSB, shift or rotate by value", "A1 RSBS, rotate right with extend", and "A1 RSBS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T1 RSB, rotate right with extend", "T1 RSB, shift or rotate by value", "T1 RSBS, rotate right with extend", and "T1 RSBS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	For the "A1 RSB, rotate right with extend", "A1 RSB, shift or rotate by value", "A1 RSBS, rotate right with extend", and "A1 RSBS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T1 RSB, rotate right with extend", "T1 RSB, shift or rotate by value", "T1 RSBS, rotate right with extend", and "T1 RSBS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”: <table> <tr> <th>stype</th><th><shift></th></tr> <tr> <td>00</td><td>LSL</td></tr> <tr> <td>01</td><td>LSR</td></tr> <tr> <td>10</td><td>ASR</td></tr> <tr> <td>11</td><td>ROR</td></tr> </table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the "A1 RSB, shift or rotate by value" and "A1 RSBS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32. For the "T1 RSB, shift or rotate by value" and "T1 RSBS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.\[N,Z,C,V\] = nzcvc;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSB, RSBS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	1	1	S	Rn				Rd				Rs				0	stype		1	Rm			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

RSBS{<c>}{<q>}{<Rd>}, <Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

RSB{<c>}{<q>}{<Rd>}, <Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”:										
<table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>		stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<Rs>	Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{R(m), shift_t, shift_n, PSTATE.C};
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, '1');
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

RSC, RSCS (immediate)

Reverse Subtract with Carry (immediate) subtracts a register value and the value of NOT (Carry flag) from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	1	1	1	S	Rn				Rd				imm12											
cond				opc																											

Encoding for the RSC variant

Applies when (S == 0)

RSC{<c>}{<q>}{<Rd>}, <Rn>, #<const>

Encoding for the RSCS variant

Applies when (S == 1)

RSCS{<c>}{<q>}{<Rd>}, <Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: • For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. • For the RSCS variant, the instruction performs an exception return, that restores <i>PSTATE</i> from SPSR_<current_mode>.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
<const>	An immediate value. See Modified immediate constants in A32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), imm32, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSC, RSCS (register)

Reverse Subtract with Carry (register) subtracts a register value and the value of NOT (Carry flag) from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	1	1	S	Rn				Rd				imm5				stype		0	Rm				
cond								opc																							

Encoding for the RSC, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

RSC{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX

Encoding for the RSC, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

RSC{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the RSCS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

RSCS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX

Encoding for the RSCS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

RSCS{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

Assembler Symbols

- <c>

See *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <Rd>

Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
 - For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
 - For the RSCS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.
- <Rn>

Is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
- <Rm>

Is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”:
stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

RSC, RSCS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value and the value of NOT (Carry flag) from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	1	1	S	Rn				Rd				Rs				0	stype		1	Rm			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

RSCS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

RSC{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{R(m), shift_t, shift_n, PSTATE.C};
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

Operational information

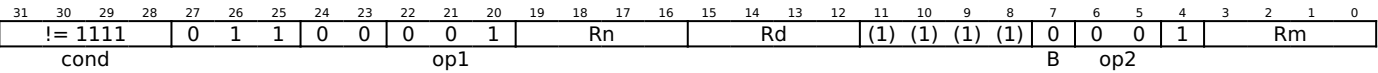
This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



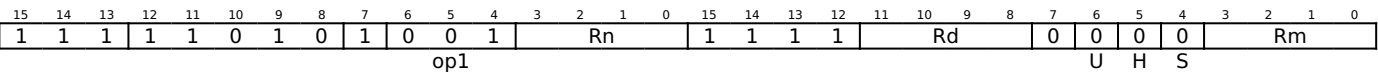
Encoding

SADD16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SADD16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[15:0]) + SInt(R(m)[15:0]);
    let sum2 : integer = SInt(R(n)[31:16]) + SInt(R(m)[31:16]);
    R(d)[15:0] = sum1[15:0];
    R(d)[31:16] = sum2[15:0];
    PSTATE.GE[1:0] = if sum1 >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if sum2 >= 0 then '11' else '00';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	0	1	Rn				Rd				(1)	(1)	(1)	(1)	1	0	0	1	Rm			
cond				op1																B				op2							

Encoding

SADD8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	0	Rn				1	1	1	1	Rd				0	0	0	0	Rm			
op1												U												H				S			

Encoding

SADD8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[7:0]) + SInt(R(m)[7:0]);
    let sum2 : integer = SInt(R(n)[15:8]) + SInt(R(m)[15:8]);
    let sum3 : integer = SInt(R(n)[23:16]) + SInt(R(m)[23:16]);
    let sum4 : integer = SInt(R(n)[31:24]) + SInt(R(m)[31:24]);
    R(d)[7:0] = sum1[7:0];
    R(d)[15:8] = sum2[7:0];
    R(d)[23:16] = sum3[7:0];
    R(d)[31:24] = sum4[7:0];
    PSTATE.GE[0] = if sum1 >= 0 then '1' else '0';
    PSTATE.GE[1] = if sum2 >= 0 then '1' else '0';
    PSTATE.GE[2] = if sum3 >= 0 then '1' else '0';
    PSTATE.GE[3] = if sum4 >= 0 then '1' else '0';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

SASX

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets *PSTATE*.GE according to the results. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	0	1	Rn				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rm			
cond				op1																B				op2							

Encoding

SASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	0	Rn				1	1	1	1	Rd				0	0	0	0	Rm			
op1												U												H				S			

Encoding

SASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = SInt(R(n)[15:0]) - SInt(R(m)[31:16]);
    let sum : integer = SInt(R(n)[31:16]) + SInt(R(m)[15:0]);
    R(d)[15:0] = diff[15:0];
    R(d)[31:16] = sum[15:0];
    PSTATE.GE[1:0] = if diff >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if sum >= 0 then '11' else '00';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

SB

Speculation Barrier is a barrier that controls speculation. For more information and details of the semantics, see *Speculation Barrier (SB)*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_SB)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	1	(0)	(0)	(0)	(0)
																opcode								option							

Encoding

SB{<q>}

Decode for this encoding

```
// No additional decoding required
```

T1
(FEAT_SB)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	1	(0)	(0)	(0)	(0)
																opc								option							

Encoding

SB{<q>}

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculationBarrier();
end;
```

SBC, SBCS (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	1	1	0	S	Rn				Rd				imm12											
cond								opc																							

Encoding for the SBC variant

Applies when (S == 0)

SBC{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the SBCS variant

Applies when (S == 1)

SBCS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	1	0	1	1	S	Rn				0	imm3				Rd				imm8							
op1																																

Encoding for the SBC variant

Applies when (S == 0)

SBC{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the SBCS variant

Applies when (S == 1)

SBCS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

<Rd>	For the "A1 SBC" and "A1 SBCS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: - For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. For the "T1 SBC" and "T1 SBCS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For the "A1 SBC" and "A1 SBCS" variants: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T1 SBC" and "T1 SBCS" variants: is the general-purpose source register, encoded in the "Rn" field.
<const>	For the "A1 SBC" and "A1 SBCS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values. For the "T1 SBC" and "T1 SBCS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(imm32), PSTATE.C);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBC, SBCS (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111								0	0	0	0	1	1	0	S	Rn				Rd				imm5				stype	0	Rm			
cond								opc																									

Encoding for the SBC, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

SBC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SBC, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

SBC{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the SBCS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

SBCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SBCS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

SBCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	Rm			Rdn		
op															

Encoding

SBC{<c>}{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())

SBCS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	1	1	S			Rn		(0)			imm3				Rd			imm2		stype			Rm
op1																															

Encoding for the SBC, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

SBC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SBC, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SBC{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

SBC<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the SBCS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

SBCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SBCS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SBCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

SBCS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> For the "A1 SBC, rotate right with extend", "A1 SBC, shift or rotate by value", "A1 SBCS, rotate right with extend", and "A1 SBCS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the SBCS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the "T2 SBC, rotate right with extend", "T2 SBC, shift or rotate by value", "T2 SBCS, rotate right with extend", and "T2 SBCS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For the "A1 SBC, rotate right with extend", "A1 SBC, shift or rotate by value", "A1 SBCS, rotate right with extend", and "A1 SBCS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For the "T2 SBC, rotate right with extend", "T2 SBC, shift or rotate by value", "T2 SBCS, rotate right with extend", and "T2 SBCS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For the "A1 SBC, rotate right with extend", "A1 SBC, shift or rotate by value", "A1 SBCS, rotate right with extend", and "A1 SBCS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T1", "T2 SBC, rotate right with extend", "T2 SBC, shift or rotate by value", "T2 SBCS, rotate right with extend", and "T2 SBCS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.

<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”:										
	<table> <tr> <th>stype</th><th><shift></th></tr> <tr> <td>00</td><td>LSL</td></tr> <tr> <td>01</td><td>LSR</td></tr> <tr> <td>10</td><td>ASR</td></tr> <tr> <td>11</td><td>ROR</td></tr> </table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the "A1 SBC, shift or rotate by value" and "A1 SBCS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32. For the "T2 SBC, shift or rotate by value" and "T2 SBCS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.										
<Rdn>	Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.										

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), PSTATE.C);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.\[N,Z,C,V\] = nzcvc;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBC, SBCS (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	1	0	S	Rn				Rd				Rs				0	stype		1	Rm			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

SBCS{<c>}{<q>}{<Rd>}, <Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

SBC{<c>}{<q>}{<Rd>}, <Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<Rs>	Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{R(m), shift_t, shift_n, PSTATE.C};
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	0	1	widthm1				Rd				lsb				1 0		1	Rn					
cond									U																						

Encoding

SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(lsb);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	0	0	Rn				0	imm3				Rd				imm2		(0)	widthm1			
op1																															

Encoding

SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(imm3::imm2);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the "A1" variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.

For the "T1" variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.

<width> Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d) = SignExtend{32}(R(n)[msbit:lsbit]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition flags are not affected. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	0	0	1	Rd				(1)	(1)	(1)	(1)	Rm				0	0	0	1	Rn			
cond				op1								Ra								op2											

Encoding

SDIV{<c>}{<q> } {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If **Ra != '1111'**, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction executes as described, with no change to its behavior and no additional side effects.
 - The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	0	0	1	Rn				(1)	(1)	(1)	(1)	Rd				1	1	1	1	Rm			
op1												Ra								op2											

Encoding

SDIV{<c>}{<q> } {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a != 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If **Ra != '1111'**, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction executes as described, with no change to its behavior and no additional side effects.
 - The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.
Overflow	

If the signed integer division $0x80000000 / 0xFFFFFFFF$ is performed, the pseudocode produces the intermediate integer result $+2^{31}$, that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to $\langle Rd \rangle$ must be the bottom 32 bits of the binary representation of $+2^{31}$. So the result of the division is $0x80000000$.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let dividend : integer = SInt(R(n));
    let divisor : integer = SInt(R(m));
    var result : integer;
    if divisor == 0 then
        result = 0;
    elsif (dividend < 0) == (divisor < 0) then
        result = Abs(dividend) DIVRM Abs(divisor); // same signs - positive result
    else
        result = -(Abs(dividend) DIVRM Abs(divisor)); // different signs - negative result
    end;
    R(d) = result[31:0];
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the *PSTATE*.GE flags. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	0	0	Rn				Rd				(1)	(1)	(1)	(1)	1	0	1	1	Rm			
cond																															

Encoding

SEL{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	0	Rn				1	1	1	1	Rd				1	0	0	0	Rm			
op1												op2																			

Encoding

SEL{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[7:0] = if PSTATE.GE[0] == '1' then R(n)[7:0] else R(m)[7:0];
    R(d)[15:8] = if PSTATE.GE[1] == '1' then R(n)[15:8] else R(m)[15:8];
    R(d)[23:16] = if PSTATE.GE[2] == '1' then R(n)[23:16] else R(m)[23:16];
    R(d)[31:24] = if PSTATE.GE[3] == '1' then R(n)[31:24] else R(m)[31:24];
end;
```

SETEND

Set Endianness writes a new value to *PSTATE.E*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	0	1	0	0	0	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)	(0)	(0)	E	(0)	0	0	0	0	0	(0)	(0)	(0)	(0)
imod												M		op		A						I		F		mode						

Encoding

```
SETEND{<q>} <endian_specifier> // (Cannot be conditional)
```

Decode for this encoding

```
let set_bigend : boolean = (E == '1');
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	0	(1)	E	(0)	(0)	(0)
op															

Encoding

```
SETEND{<q>} <endian_specifier> // (Not permitted in IT block)
```

Decode for this encoding

```
let set_bigend : boolean = (E == '1');
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <endian_specifier> Is the endianness to be selected, and the value to be set in PSTATE.E, encoded in “E”:

E	<endian_specifier>
0	LE
1	BE

Operation

```
EncodingSpecificOperations();
AArch32_CheckSETENDEnabled();
PSTATE.E = if set_bigend then '1' else '0';
```

SETPAN

Set Privileged Access Never writes a new value to *PSTATE*.PAN.
This instruction is available only in privileged mode and it is a NOP when executed in User mode.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_PAN)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	1	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	imm1	(0)	0	0	0	0	(0)	(0)	(0)	(0)

Encoding

```
SETPAN{<q>} #<imm> // (Cannot be conditional)
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAN) then Undefined(); end;  
let value : bit = imm1;
```

T1
(FEAT_PAN)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	0	0	(1)	imm1	(0)	(0)	(0)

Encoding

```
SETPAN{<q>} #<imm> // (Not permitted in IT block)
```

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;  
if !IsFeatureImplemented(FEAT_PAN) then Undefined(); end;  
let value : bit = imm1;
```

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <imm> Is the unsigned immediate 0 or 1, encoded in the "imm1" field.

Operation

```
EncodingSpecificOperations();  
if PSTATE.EL != EL0 then  
    PSTATE.PAN = value;  
end;
```

SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait For Event and Send Event*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	1	0	0
cond				R								imm4				imm12															

Encoding

SEV{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0
hint															

Encoding

SEV{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	0
hint																option															

Encoding

SEV{<c>}.W

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c>See *Standard assembler syntax fields*.
- <q>See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
end;
```

SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	1	0	1
cond				R								imm4				imm12															

Encoding

SEVL{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0
hint															

Encoding

SEVL{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	1
hint																option															

Encoding

SEVL{<c>}.W

Decode for this encoding

// No additional decoding required

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).

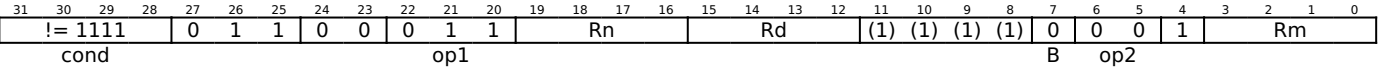
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEventLocal();
end;
```

SHADD16

Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



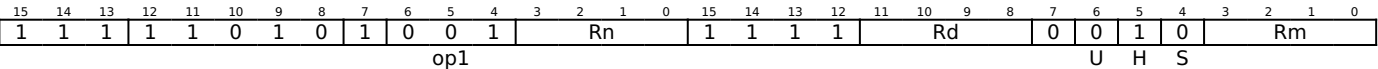
Encoding

```
SHADD16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
SHADD16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[15:0]) + SInt(R(m)[15:0]);
    let sum2 : integer = SInt(R(n)[31:16]) + SInt(R(m)[31:16]);
    R(d)[15:0] = sum1[16:1];
    R(d)[31:16] = sum2[16:1];
end;
```

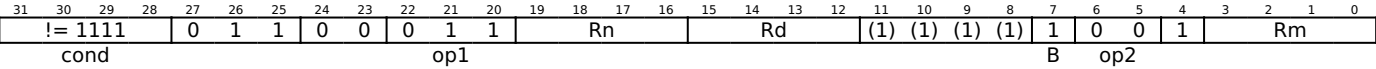
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



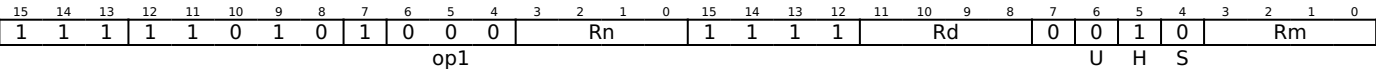
Encoding

```
SHADD8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
SHADD8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[7:0]) + SInt(R(m)[7:0]);
    let sum2 : integer = SInt(R(n)[15:8]) + SInt(R(m)[15:8]);
    let sum3 : integer = SInt(R(n)[23:16]) + SInt(R(m)[23:16]);
    let sum4 : integer = SInt(R(n)[31:24]) + SInt(R(m)[31:24]);
    R(d)[7:0] = sum1[8:1];
    R(d)[15:8] = sum2[8:1];
    R(d)[23:16] = sum3[8:1];
    R(d)[31:24] = sum4[8:1];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHASX

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	1	1	Rn				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rm			
cond				op1																B				op2							

Encoding

SHASX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	0	Rn				1	1	1	1	Rd				0	0	1	0	Rm			
op1												U												H				S			

Encoding

SHASX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = SInt(R(n)[15:0]) - SInt(R(m)[31:16]);
    let sum : integer = SInt(R(n)[31:16]) + SInt(R(m)[15:0]);
    R(d)[15:0] = diff[16:1];
    R(d)[31:16] = sum[16:1];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHSAX

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	1	1	Rn				Rd				(1)	(1)	(1)	(1)	0	1	0	1	Rm			
cond				op1												B				op2											

Encoding

SHSAX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	1	0	Rn				1	1	1	1	Rd				0	0	1	0	Rm			
op1												U												H				S			

Encoding

SHSAX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = SInt(R(n)[15:0]) + SInt(R(m)[31:16]);
    let diff : integer = SInt(R(n)[31:16]) - SInt(R(m)[15:0]);
    R(d)[15:0] = sum[16:1];
    R(d)[31:16] = diff[16:1];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	1	1	Rn				Rd				(1)	(1)	(1)	(1)	0	1	1	1	Rm			
cond				op1																B				op2							

Encoding

SHSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	1	Rn				1	1	1	1	Rd				0	0	1	0	Rm			
								op1												U				H				S			

Encoding

SHSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[15:0]) - SInt(R(m)[15:0]);
    let diff2 : integer = SInt(R(n)[31:16]) - SInt(R(m)[31:16]);
    R(d)[15:0] = diff1[16:1];
    R(d)[31:16] = diff2[16:1];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHSUB8

Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	1	1	Rn				Rd				(1)	(1)	(1)	(1)	1	1	1	1	Rm			
cond				op1																B				op2							

Encoding

SHSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	0	Rn				1	1	1	1	Rd				0	0	1	0	Rm			
								op1												U				H				S			

Encoding

SHSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[7:0]) - SInt(R(m)[7:0]);
    let diff2 : integer = SInt(R(n)[15:8]) - SInt(R(m)[15:8]);
    let diff3 : integer = SInt(R(n)[23:16]) - SInt(R(m)[23:16]);
    let diff4 : integer = SInt(R(n)[31:24]) - SInt(R(m)[31:24]);
    R(d)[7:0] = diff1[8:1];
    R(d)[15:8] = diff2[8:1];
    R(d)[23:16] = diff3[8:1];
    R(d)[31:24] = diff4[8:1];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SMC

Secure Monitor Call causes a Secure Monitor Call exception. For more information, see [Secure Monitor Call \(SMC\) exception](#).

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in User mode.

If the values of [HCR.TSC](#) and [SCR.SCD](#) are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception that is taken to EL3. When EL3 is using AArch32 this exception is taken to Monitor mode. When EL3 is using AArch64, it is the SCR_EL3.SMD bit, rather than the [SCR.SCD](#) bit, that can change the effect of executing an SMC instruction.

If the value of [HCR.TSC](#) is 1, execution of an SMC instruction in a Non-secure EL1 mode generates an exception that is taken to EL2, regardless of the value of [SCR.SCD](#). When EL2 is using AArch32, this is a Hyp Trap exception that is taken to Hyp mode. For more information, see [HCR.TSC](#).

If the value of [HCR.TSC](#) is 0 and the value of [SCR.SCD](#) is 1, the SMC instruction is:

- UNDEFINED in Non-secure state.
- CONSTRAINED UNPREDICTABLE if executed in Secure state at EL1 or higher.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	1	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	0	1	1	1	imm4				
cond				opc								imm12																			

Encoding

SMC{<c>}{<q>}{#}<imm4>

Decode for this encoding

```
// imm4 is for assembly/disassembly only and is ignored by hardware
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0												
1	1	1	1	0	1	1	1	1	1	1	1	imm4				1	0	0	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)												
o1												o2																imm12															

Encoding

SMC{<c>}{<q>}{#}<imm4>

Decode for this encoding

```
// imm4 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c>

See [Standard assembler syntax fields](#).
- <q>

See [Standard assembler syntax fields](#).
- <imm4>

Is a 4-bit unsigned immediate value, in the range 0 to 15, encoded in the "imm4" field. This is ignored by the PE. The Secure Monitor Call exception handler (Secure Monitor code) can use this value to determine what service is being requested, but Arm does not recommend this.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations\(\);

    AArch32\_CheckForSMCUndefinedOrTrap\(\);

    if !ELUsingAArch32\(EL3\) then
        if SCR\_EL3\(\).SMD == '1' then
            // SMC disabled.
            Undefined\(\);
        end;
    else
        if SCR\(\).SCD == '1' then
            // SMC disabled
            if CurrentSecurityState\(\) == SS\_Secure then
                // Executes either as a NOP or UNALLOCATED.
                let c : Constraint = ConstrainUnpredictable\(Unpredictable\_SMD\);
                assert c IN {Constraint\_NOP, Constraint\_UNDEF};
                if c == Constraint\_NOP then ExecuteAsNOP\(\); end;
            end;
            Undefined\(\);
        end;
    end;

    if !ELUsingAArch32\(EL3\) then
        AArch64\_CallSecureMonitor(Zeros{16});
    else
        AArch32\_TakeSMCException\(\);
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `SCR.SCD == '1' && CurrentSecurityState() == SS_Secure`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets *PSTATE.Q* to 1. It is not possible for overflow to occur during the multiplication.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0	0	0	Rd				Ra				Rm				1	M	N	0	Rn			
cond									opc																						

Encoding for the SMLABB variant

Applies when (M == 0 && N == 0)

SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLABT variant

Applies when (M == 1 && N == 0)

SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLATB variant

Applies when (M == 0 && N == 1)

SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLATT variant

Applies when (M == 1 && N == 1)

SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	1	0	0	0	1	Rn				!= 1111				Rd				0	0	N	M	Rm							
op1												Ra																							

Encoding for the SMLABB variant

Applies when (N == 0 && M == 0)

SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLABT variant

Applies when (N == 0 && M == 1)

SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLATB variant

Applies when (N == 1 && M == 0)

SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLATT variant

Applies when (N == 1 && M == 1)

SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMULBB, SMULBT, SMULTB, SMULTT"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : bits(16) = R(n)[n_index*16];
    let operand2 : bits(16) = R(m)[m_index*16];
    let result : integer = SInt(operand1) * SInt(operand2) + SInt(R(a));
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLAD, SMLADX

Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand. Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets *PSTATE.Q* to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	0 0 0			Rd				!= 1111				Rm				0	0	M	1	Rn			
cond				op1								Ra																			

Encoding for the SMLAD variant

Applies when (M == 0)

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLADX variant

Applies when (M == 1)

SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUAD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	1	1	0	0	1	0	Rn				!= 1111				Rd				0		0	0	M	Rm			
op1												Ra																				

Encoding for the SMLAD variant

Applies when (M == 0)

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLADX variant

Applies when (M == 1)

SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUAD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 + product2 + SInt(R(a));
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLAL, SMLALS

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value. In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	1	1	S	RdHi				RdLo				Rm				1	0	0	1	Rn			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

SMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the Not flag setting variant

Applies when (S == 0)

SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	1	0	0	Rn				RdLo				RdHi				0	0	0	0	Rm			
op1												op2																			

Encoding

SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = SInt(R(n)) * SInt(R(m)) + SInt(R(dHi) :: R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value. Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64} . It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	0	0	RdHi				RdLo				Rm			1	M	N	0	Rn				
cond									opc																						

Encoding for the SMLALBB variant

Applies when (M == 0 && N == 0)

SMLALBB{<c>} {<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALBT variant

Applies when (M == 1 && N == 0)

SMLALBT{<c>} {<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALTB variant

Applies when (M == 0 && N == 1)

SMLALTB{<c>} {<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALTT variant

Applies when (M == 1 && N == 1)

SMLALTT{<c>} {<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	1	0	0	Rn				RdLo				RdHi				1	0	N	M	Rm			
op1																															

Encoding for the SMLALBB variant

Applies when (N == 0 && M == 0)

SMLALBB{<c>} {<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALBT variant

Applies when (N == 0 && M == 1)

SMLALBT{<c>} {<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALTB variant

Applies when (N == 1 && M == 0)

SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALTT variant

Applies when (N == 1 && M == 1)

SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	For the "A1 SMLALBB", "A1 SMLALBT", "A1 SMLALTB", and "A1 SMLALTT" variants: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field. For the "T1 SMLALBB", "T1 SMLALBT", "T1 SMLALTB", and "T1 SMLALTT" variants: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.
<Rm>	For the "A1 SMLALBB", "A1 SMLALBT", "A1 SMLALTB", and "A1 SMLALTT" variants: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. For the "T1 SMLALBB", "T1 SMLALBT", "T1 SMLALTB", and "T1 SMLALTT" variants: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <x>), encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : bits(16) = R(n)[n_index*16];
    let operand2 : bits(16) = R(m)[m_index*16];
    let result : integer = SInt(operand1) * SInt(operand2) + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLALD, SMLALDX

Signed Multiply Accumulate Long Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand. Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2⁶⁴.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	1 0 0			RdHi				RdLo				Rm				0	0	M	1	Rn			
cond									op1																						

Encoding for the SMLALD variant

Applies when (M == 0)

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALDX variant

Applies when (M == 1)

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	1	0	0	Rn				RdLo				RdHi				1	1	0	M	Rm			
									op1																						

Encoding for the SMLALD variant

Applies when (M == 0)

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALDX variant

Applies when (M == 1)

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 + product2 + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

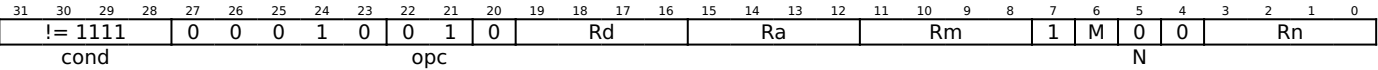
SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets *PSTATE.Q* to 1. No overflow can occur during the multiplication.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



Encoding for the SMLAWB variant

Applies when (M == 0)

SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLAWT variant

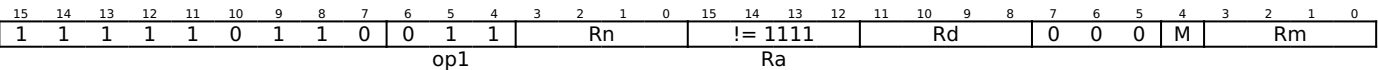
Applies when (M == 1)

SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_high : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the SMLAWB variant

Applies when (M == 0)

SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLAWT variant

Applies when (M == 1)

SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMULWB, SMULWT"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_high : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(16) = if m_high then R(m)[31:16] else R(m)[15:0];
    let result : integer = SInt(R(n)) * SInt(operand2) + (SInt(R(a)) << 16);
    R(d) = result[47:16];
    if (result >> 16) != SInt(R(d)) then // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLSD, SMLSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets *PSTATE.Q* to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	0 0 0 0			Rd				!= 1111				Rm				0	1	M	1	Rn			
cond				op1								Ra																			

Encoding for the SMLSD variant

Applies when (M == 0)

SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLSDX variant

Applies when (M == 1)

SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUSD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	1	0	0	Rn				!= 1111				Rd				0	0	0	M	Rm			
op1												Ra																			

Encoding for the SMLSD variant

Applies when (M == 0)

SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLSDX variant

Applies when (M == 1)

SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUSD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = (product1 - product2) + SInt(R(a));
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLS LD, SMLS LDx

Signed Multiply Subtract Long Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64} .

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	1 0 0			RdHi				RdLo				Rm				0	1	M	1	Rn			
cond									op1																						

Encoding for the SMLS LD variant

Applies when (M == 0)

SMLS LD{<c>}{<q>}<RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLS LDx variant

Applies when (M == 1)

SMLS LDx{<c>}{<q>}<RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	1	0	1	Rn				RdLo				RdHi				1	1	0	M	Rm			
									op1																						

Encoding for the SMLS LD variant

Applies when (M == 0)

SMLS LD{<c>}{<q>}<RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLS LDx variant

Applies when (M == 1)

SMLS LDx{<c>}{<q>}<RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The value in the destination register is UNKNOWN.
- For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = (product1 - product2) + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

SMMLA, SMMLAR

Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	1	0	1	Rd				!= 1111				Rm				0	0	R	1	Rn			
cond				op1								Ra																			

Encoding for the SMMLA variant

Applies when (R == 0)

SMMLA{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMMLAR variant

Applies when (R == 1)

SMMLAR{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMMUL"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	1	0	1	Rn				!= 1111				Rd				0	0	0	R	Rm			
op1												Ra																			

Encoding for the SMMLA variant

Applies when (R == 0)

SMMLA{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMMLAR variant

Applies when (R == 1)

SMMLAR{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMMUL"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : integer = (SInt(R(a)) << 32) + SInt(R(n)) * SInt(R(m));
    if round then result = result + 0x80000000; end;
    R(d) = result[63:32];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMMLS, SMMLSR

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the result of the subtraction before the high word is extracted.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	1	0	1	Rd				Ra				Rm				1	1	R	1	Rn			
cond									op1																						

Encoding for the SMMLS variant

Applies when (R == 0)

SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMMLSR variant

Applies when (R == 1)

SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	1	1	0	Rn				Ra				Rd				0	0	0	R	Rm			
op1																															

Encoding for the SMMLS variant

Applies when (R == 0)

SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMMLSR variant

Applies when (R == 1)

SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : integer = (SInt(R(a)) << 32) - SInt(R(n)) * SInt(R(m));
    if round then result = result + 0x80000000; end;
    R(d) = result[63:32];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMMUL, SMMULR

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	1 0 1			Rd				1 1 1 1				Rm				0 0		R	1	Rn			
cond				op1								Ra																			

Encoding for the SMMUL variant

Applies when (R == 0)

SMMUL{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Encoding for the SMMULR variant

Applies when (R == 1)

SMMULR{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let round : boolean = (R == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	1	0	1	Rn				1	1	1	1	Rd				0	0	0	R	Rm			
op1												Ra																			

Encoding for the SMMUL variant

Applies when (R == 0)

SMMUL{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Encoding for the SMMULR variant

Applies when (R == 1)

SMMULR{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let round : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : integer = SInt(R(n)) * SInt(R(m));
    if round then result = result + 0x80000000; end;
    R(d) = result[63:32];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMUAD, SMUADX

Signed Dual Multiply Add performs two signed 16 x 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets *PSTATE.Q* to 1 if the addition overflows. The multiplications cannot overflow.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	0	0	0	Rd				1	1	1	1	Rm				0	0	M	1	Rn			
cond				op1								Ra																			

Encoding for the SMUAD variant

Applies when (M == 0)

SMUAD{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Encoding for the SMUADX variant

Applies when (M == 1)

SMUADX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	0	1	0	Rn				1	1	1	1	Rd				0	0	0	M	Rm			
op1												Ra																			

Encoding for the SMUAD variant

Applies when (M == 0)

SMUAD{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Encoding for the SMUADX variant

Applies when (M == 1)

SMUADX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 + product2;
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	1	1	0	Rd				(0) (0) (0) (0)				Rm				1	M	N	0	Rn			
cond				opc								Ra																			

Encoding for the SMULBB variant

Applies when (M == 0 && N == 0)

SMULBB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULBT variant

Applies when (M == 1 && N == 0)

SMULBT{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULTB variant

Applies when (M == 0 && N == 1)

SMULTB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULTT variant

Applies when (M == 1 && N == 1)

SMULTT{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	0	0	1	Rn				1	1	1	1	Rd				0	0	N	M	Rm			
op1												Ra																			

Encoding for the SMULBB variant

Applies when (N == 0 && M == 0)

SMULBB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULBT variant

Applies when (N == 0 && M == 1)

SMULBT{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULTB variant

Applies when (N == 1 && M == 0)

SMULTB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULTT variant

Applies when (N == 1 && M == 1)

SMULTT{<c>}{<q>}{<Rd>, <Rn>, <Rm>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : bits(16) = R(n)[n_index*16];
    let operand2 : bits(16) = R(m)[m_index*16];
    let result : integer = SInt(operand1) * SInt(operand2);
    R(d) = result[31:0];
    // Signed overflow cannot occur
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULL, SMULLS

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.
In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	1	0	S	RdHi				RdLo				Rm				1	0	0	1	Rn			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

SMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the Not flag setting variant

Applies when (S == 0)

SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	0	0	0	Rn				RdLo				RdHi				0	0	0	0	Rm			
op1																												op2			

Encoding

SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q>	See <i>Standard assembler syntax fields</i> .
<RdLo>	Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = SInt(R(n)) * SInt(R(m));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULWB, SMULWT

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	0	1	0	Rd				(0)	(0)	(0)	(0)	Rm				1	M	1	0	Rn			
cond								opc				Ra								N											

Encoding for the SMULWB variant

Applies when (M == 0)

SMULWB{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Encoding for the SMULWT variant

Applies when (M == 1)

SMULWT{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_high : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	0	1	1	Rn				1	1	1	1	Rd				0	0	0	M	Rm			
								op1				Ra																			

Encoding for the SMULWB variant

Applies when (M == 0)

SMULWB{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Encoding for the SMULWT variant

Applies when (M == 1)

SMULWT{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_high : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(16) = if m_high then R(m)[31:16] else R(m)[15:0];
    let product : integer = SInt(R(n)) * SInt(operand2);
    R(d) = product[47:16];
    // Signed overflow cannot occur
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMUSD, SMUSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow cannot occur.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	0	0	0	Rd				1	1	1	1	Rm				0	1	M	1	Rn			
cond				op1								Ra																			

Encoding for the SMUSD variant

Applies when (M == 0)

SMUSD{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Encoding for the SMUSDX variant

Applies when (M == 1)

SMUSDX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	1	0	0	Rn				1	1	1	1	Rd				0	0	0	M	Rm			
op1												Ra																			

Encoding for the SMUSD variant

Applies when (M == 0)

SMUSD{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Encoding for the SMUSDX variant

Applies when (M == 1)

SMUSDX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 - product2;
    R(d) = result[31:0];
    // Signed overflow cannot occur
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRS, SRSDA, SRSDb, SRSIA, SRSIB

Store Return State stores the LR_<current_mode> and SPSR_<current_mode> to the stack of a specified mode. For information about memory accesses see *Memory accesses*.

SRS is UNDEFINED in Hyp mode.

SRS is CONSTRAINED UNPREDICTABLE if it is executed in User or System mode, or if the specified mode is any of the following:

- Not implemented.
- A mode that AArch32 PE modes does not show.
- Hyp mode.
- Monitor mode, if the SRS instruction is executed in Non-secure state.

If EL3 is using AArch64 and an SRS instruction that is executed in a Secure EL1 mode specifies Monitor mode, it is trapped to EL3.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	1	W	0	(1)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	mode				
									S				L	Rn				op													

Encoding for the Decrement After variant

Applies when (P == 0 && U == 0)

SRSDA{<c>}{<q>} SP{!}, #<mode>

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0)

SRSDb{<c>}{<q>} SP{!}, #<mode>

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

SRS{IA}{<c>}{<q>} SP{!}, #<mode>

Encoding for the Increment Before variant

Applies when (P == 1 && U == 1)

SRSIB{<c>}{<q>} SP{!}, #<mode>

Decode for all variants of this encoding

```
let wback : boolean = (W == '1');
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	0	0	0	0	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode						
opc							L				Rn					P	M																

Encoding

SRSDb{<c>}{<q>} SP{!}, #<mode>

Decode for this encoding

```
let wback : boolean = (W == '1');
let increment : boolean = FALSE;
let wordhigher : boolean = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				
opc							L				Rn					P	M														

Encoding

SRS{IA}{<c>}{<q>} SP{!}, #<mode>

Decode for this encoding

```
let wback : boolean = (W == '1');
let increment : boolean = TRUE;
let wordhigher : boolean = FALSE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [SRS \(T32\)](#) and [SRS \(A32\)](#).

Assembler Symbols

<c>	For the "Decrement After", "Decrement Before", "Increment After", and "Increment Before" variants: see Standard assembler syntax fields . <c> must be AL or omitted. For the "T1" and "T2" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<mode>	Is the number of the mode whose Banked SP is used as the base register, encoded in the "mode" field. For details of PE modes and their numbers see AArch32 state PE modes .
IA	For the "Increment After" variant: is an optional suffix to indicate the Increment After variant. For the "T2" variant: is an optional suffix for the Increment After form.

SRSFA, SRSEA, SRSFD, and SRSED are pseudo-instructions for SRSIB, SRSIA, SRSDb, and SRSDA respectively, referring to their use for pushing data onto Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```
if CurrentInstrSet\(\) == InstrSet\_A32 then
  if ConditionPassed\(\) then
    EncodingSpecificOperations\(\);
    if PSTATE.EL == EL2 then          // UNDEFINED at EL2
      Undefined\(\);
    end;

    // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
    // to be security holes
    if PSTATE.M IN {M32\_User,M32\_System} then
      UnpredictableProcedure\(\);
    elsif mode == M32\_Hyp then          // Check for attempt to access Hyp mode SP
      UnpredictableProcedure\(\);
    elsif mode == M32\_Monitor then      // Check for attempt to access Monitor mode SP
      if !HaveEL\(EL3\) || CurrentSecurityState\(\) != SS\_Secure then
        UnpredictableProcedure\(\);
      elsif !ELUsingAArch32\(EL3\) then
        AArch64\_MonitorModeTrap\(\);
      end;
    elsif BadMode\(mode\) then
      UnpredictableProcedure\(\);
    end;

    let base : bits(32) = Rmode\(13,mode\);
    var address : bits(32) = if increment then base else base-8;
    if wordhigher then address = address+4; end;
    MemA{32}\(address\) = LR\(\);
    MemA{32}\(address+4\) = SPSR\_curr\(\);
    if wback then Rmode\(13,mode\) = if increment then base+8 else base-8; end;
  end;
else
  if ConditionPassed\(\) then
    EncodingSpecificOperations\(\);
    if PSTATE.EL == EL2 then          // UNDEFINED at EL2
      Undefined\(\);
    end;

    // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
    // to be security holes
    if PSTATE.M IN {M32\_User,M32\_System} then
      UnpredictableProcedure\(\);
    elsif mode == M32\_Hyp then          // Check for attempt to access Hyp mode SP
      UnpredictableProcedure\(\);
    elsif mode == M32\_Monitor then      // Check for attempt to access Monitor mode SP
      if !HaveEL\(EL3\) || CurrentSecurityState\(\) != SS\_Secure then
        UnpredictableProcedure\(\);
      elsif !ELUsingAArch32\(EL3\) then
        AArch64\_MonitorModeTrap\(\);
      end;
    elsif BadMode\(mode\) then
      UnpredictableProcedure\(\);
    end;

    let base : bits(32) = Rmode\(13,mode\);
    var address : bits(32) = if increment then base else base-8;
    if wordhigher then address = address+4; end;
    MemA{32}\(address\) = LR\(\);
    MemA{32}\(address+4\) = SPSR\_curr\(\);
    if wback then Rmode\(13,mode\) = if increment then base+8 else base-8; end;
  end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If [PSTATE.M](#) IN {[M32_User](#),[M32_System](#)}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If mode == [M32_Hyp](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If mode == [M32_Monitor](#) && ([!HaveEL\(EL3\)](#) || [CurrentSecurityState\(\)](#) != [SS_Secure](#)), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If [BadMode\(mode\)](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction stores to the stack of the mode in which it is executed.

- The instruction stores to an UNKNOWN address, and if the instruction specifies writeback then any general-purpose register that can be accessed from the current Exception level without a privilege violation becomes UNKNOWN.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

This instruction sets *PSTATE.Q* to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	1	sat_imm		Rd						imm5				sh	0	1	Rn					
cond										U																					

Encoding for the Arithmetic shift right variant

Applies when (sh == 1)

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Encoding for the Logical shift left variant

Applies when (sh == 0)

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(sh::'0', imm5);
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	0	sh	0	Rn				0	imm3				Rd				imm2		(0)	sat imm			

Encoding for the Arithmetic shift right variant

Applies when (sh == 1 && !(imm3 == 000 && imm2 == 00))

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Encoding for the Logical shift left variant

Applies when (sh == 0)

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
if sh == '1' && (imm3::imm2) == '0000' then See("SSAT16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(sh::'0', imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the bit position for saturation, in the range 1 to 32, encoded in the "sat_imm" field as <imm>-1.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<amount>	For the "A1 Arithmetic shift right" variant: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32. For the "A1 Logical shift left" variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field. For the "T1 Arithmetic shift right" variant: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

For the "T1 Logical shift left" variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let operand : bits(32) = Shift{}(R(n), shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
  let (result, sat) : (bits(saturate_to), boolean) = SignedSatQ{saturate_to}(SInt(operand));
  R(d) = SignExtend{32}(result);
  if sat then
    PSTATE.Q = '1';
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.
This instruction sets *PSTATE.Q* to 1 if the operation saturates.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	1	0	sat imm			Rd			(1)	(1)	(1)	(1)	0	0	1	1	Rn					
cond									U																						

Encoding

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	0	1	0	Rn				0	0	0	0	Rd				0	0	(0)	(0)	sat imm			
op1												imm3				imm2															

Encoding

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <imm> Is the bit position for saturation, in the range 1 to 16, encoded in the "sat_imm" field as <imm>-1.
- <Rn> Is the general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result1, sat1) : (bits(saturate_to), boolean) = SignedSatQ{saturate_to}(SInt(R(n)[15:0]));
    let (result2, sat2) : (bits(saturate_to), boolean) = SignedSatQ{saturate_to}(SInt(R(n)[31:16]));
    R(d)[15:0] = SignExtend{16}(result1);
    R(d)[31:16] = SignExtend{16}(result2);
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```

SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets *PSTATE*.GE according to the results. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	0	1	Rn				Rd				(1)	(1)	(1)	(1)	0	1	0	1	Rm			
cond				op1																B				op2							

Encoding

SSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	1	0	Rn				1	1	1	1	Rd				0	0	0	0	Rm			
op1												U H S																			

Encoding

SSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = SInt(R(n)[15:0]) + SInt(R(m)[31:16]);
    let diff : integer = SInt(R(n)[31:16]) - SInt(R(m)[15:0]);
    R(d)[15:0] = sum[15:0];
    R(d)[31:16] = diff[15:0];
    PSTATE.GE[1:0] = if sum >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if diff >= 0 then '11' else '00';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SSBB

Speculative Store Bypass Barrier is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions. For more information and details of the semantics, see *Speculative Store Bypass Barrier (SSBB)*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	0	0	0	0					
																								opcode							option					

Encoding

SSBB{<q>}

Decode for this encoding

```
// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	0	0	0	0
																opc							option								

Encoding

SSBB{<q>}

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

Operation

```
EncodingSpecificOperations();
SpeculativeStoreBypassBarrierToVA();
```

SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	0	1	Rn				Rd				(1)	(1)	(1)	(1)	0	1	1	1	Rm			
cond				op1																B				op2							

Encoding

SSUB16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	1	Rn				1	1	1	1	Rd				0	0	0	0	Rm			
op1												U												H				S			

Encoding

SSUB16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[15:0]) - SInt(R(m)[15:0]);
    let diff2 : integer = SInt(R(n)[31:16]) - SInt(R(m)[31:16]);
    R(d)[15:0] = diff1[15:0];
    R(d)[31:16] = diff2[15:0];
    PSTATE.GE[1:0] = if diff1 >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if diff2 >= 0 then '11' else '00';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SSUB8

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	0	0	1	Rn				Rd				(1)	(1)	(1)	(1)	1	1	1	1	Rm			
cond				op1																B				op2							

Encoding

SSUB8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	0	Rn				1	1	1	1	Rd				0	0	0	0	Rm			
op1												U H S																			

Encoding

SSUB8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[7:0]) - SInt(R(m)[7:0]);
    let diff2 : integer = SInt(R(n)[15:8]) - SInt(R(m)[15:8]);
    let diff3 : integer = SInt(R(n)[23:16]) - SInt(R(m)[23:16]);
    let diff4 : integer = SInt(R(n)[31:24]) - SInt(R(m)[31:24]);
    R(d)[7:0] = diff1[7:0];
    R(d)[15:8] = diff2[7:0];
    R(d)[23:16] = diff3[7:0];
    R(d)[31:24] = diff4[7:0];
    PSTATE.GE[0] = if diff1 >= 0 then '1' else '0';
    PSTATE.GE[1] = if diff2 >= 0 then '1' else '0';
    PSTATE.GE[2] = if diff3 >= 0 then '1' else '0';
    PSTATE.GE[3] = if diff4 >= 0 then '1' else '0';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

STC

Store data to System register calculates an address from a base register value and an immediate offset, and stores a word from the [DBGDTRRXint](#) System register to memory. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses, see [Memory accesses](#).

In an implementation that includes EL2, the permitted STC access to [DBGDTRRXint](#) can be trapped to Hyp mode, meaning that an attempt to execute an STC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [HDCR.TDA](#).

For simplicity, the STC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111				1	1	0	P	U	0	W	0	Rn				0	1	0	1	1	1	1	0	imm8									
cond								D				L				CRd				cp15													

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STC{<c>}{<q>} p14, c5, [<Rn>{, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

STC{<c>}{<q>} p14, c5, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STC{<c>}{<q>} p14, c5, [<Rn>, #<+/-><imm>]!

Encoding for the Unindexed variant

Applies when (P == 0 && U == 1 && W == 0)

STC{<c>}{<q>} p14, c5, [<Rn>], <option>

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend()(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If n == 15 && wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in [Using R15](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	0	W	0	Rn			0	1	0	1	1	1	1	0	imm8								
								D				L				CRd				cp15											

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STC{<c>}{<q>} p14, c5, [<Rn>{, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
STC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
STC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>!
```

Encoding for the Unindexed variant

Applies when (P == 0 && U == 1 && W == 0)

```
STC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in [Using R15](#).

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rn>	For the "A1 Offset" and "A1 Unindexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "A1 Post-indexed", "A1 Pre-indexed", "T1 Offset", "T1 Post-indexed", "T1 Pre-indexed", and "T1 Unindexed" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1"> <thead> <tr> <th>U</th><th>+/-</th></tr> </thead> <tbody> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </tbody> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.						
<option>	Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);

    // System register read from DBGDTRRXint.
    AArch32_SysRegRead(cp, ThisInstr(), address[31:0]);
    if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

STL

Store-Release Word stores a word from a register to memory. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	0	0	Rn				(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	0	0	1	Rt			
cond				size				L				xRd				ex				ord											

Encoding

STL{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	1	0	1	0	(1)	(1)	(1)	(1)
op0				L				Rt2				op				sz				Rd											

Encoding

STL{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    MemO{32}(address) = R(t);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

STLB

Store-Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	0	0	Rn				(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	0	0	1	Rt			
cond				size				L				xRd				ex				ord											

Encoding

STLB{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	1	0	0	0	(1)	(1)	(1)	(1)
op0				L												Rt2				op				sz				Rd			

Encoding

STLB{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    MemO{8}(address) = R(t)[7:0];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

STLEX

Store-Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	0	0	Rn				Rd				(1)	(1)	1	0	1	0	0	1	Rt			
cond				size								L								ex				ord							

Encoding

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d == t**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If **d == n**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	1	1	1	0	Rd			
				op0								L								Rt2				op				sz			

Encoding

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d == t**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If **d == n**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemO{32}(address) = R(t);
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLEXB

Store-Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	0	0	Rn				Rd				(1)	(1)	1	0	1	0	0	1	Rt			
cond				size								L								ex				ord							

Encoding

STLEXB{<c>}{<q>}<Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d == t**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If **d == n**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	1	1	0	0	Rd			
				op0								L								Rt2				op				sz			

Encoding

STLEXB{<c>}{<q>}<Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d == t**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If **d == n**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemO{8}(address) = R(t)[7:0];
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLEXD

Store-Release Exclusive Doubleword stores a doubleword from two registers to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	1	0	Rn				Rd				(1)	(1)	1	0	1	0	0	1	Rt			
cond				size								L	ex ord																		

Encoding

STLEXD{<op>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
if d == 15 || Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `Rt<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in [Using R15](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				Rt2				1	1	1	1	Rd						
op0												L																	op		sz			

Encoding

STLEXD{<op>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory.
<Rt>	For the "A1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14. For the "T1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For the "A1" variant: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>. For the "T1" variant: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    let value : bits(64) = if BigEndian(AccessType_GPR) then R(t)::R(t2) else R(t2)::R(t);
    if AArch32.ExclusiveMonitorsPass(address, 8) then
        MemO{64}(address) = value;
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

STLEXH

Store-Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0	0	0	1	1	1	1	0	Rn				Rd				(1)	(1)	1	0	1	0	0	1	Rt							
cond				size								L								(1)		(1)		1		0		1		0		0		1	

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemO{16}(address) = R(t)[15:0];
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLH

Store-Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	1	0	Rn				(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	0	0	1	Rt			
cond				size				L				xRd				ex				ord											

Encoding

STLH{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	1	0	0	1	(1)	(1)	(1)	(1)
op0				L												Rt2				op				sz				Rd			

Encoding

STLH{<c>}{<q>}<Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    MemO{16}(address) = R(t)[15:0];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

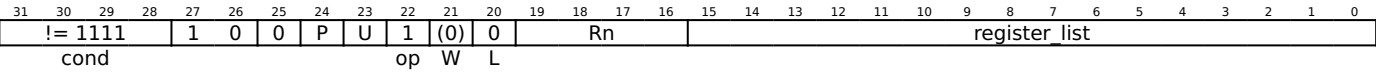
STM (User registers)

In an EL1 mode other than System mode, Store Multiple (User registers) stores multiple User mode registers to consecutive memory locations using an address from a base register. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Store Multiple (User registers) is UNDEFINED in Hyp mode, and CONSTRAINED UNPREDICTABLE in User or System modes.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#).

A1



Encoding

STM{<amode>}{<c>}{<q>} <Rn>, <registers>^

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<amode>	is one of:
DA	Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
ED	Empty Descending. For this instruction, a synonym for DA.
DB	Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
FD	Full Descending. For this instruction, a synonym for DB.
IA	Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
EA	Empty Ascending. For this instruction, a synonym for IA.
IB	Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.
FA	Full Ascending. For this instruction, a synonym for IB.
<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<registers>	Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the STM instruction. The registers are stored with the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address. See also Encoding of lists of general-purpose registers and the PC .

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        Undefined();
    elsif PSTATE.M IN {M32_User, M32_System} then
        UnpredictableProcedure();
    else
        let length : integer = 4*BitCount(registers);
        var address : bits(32) = if increment then R(n) else R(n)-length;
        if wordhigher then address = address+4; end;
        for i = 0 to 14 do
            if registers[i] == '1' then // Store User mode register
                MemS{32}(address) = Rmode(i, M32_User);
                address = address + 4;
            end;
        end;
        if registers[15] == '1' then
            MemS{32}(address) = PCStoreValue();
        end;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User, M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STM, STMIA, STMEA

Store Multiple (Increment After, Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#). For details of related system instructions see [STM \(User registers\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	0	0	0	1	0	W	0	Rn				register_list															
cond				P				U	op	L																					

Encoding

```
STM{IA}{<c>}{<q> <Rn>!, <registers> // (Preferred syntax)
```

```
STMEA{<c>}{<q> <Rn>!, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in [Using R15](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	Rn			register_list							
L															

Encoding

```
STM{IA}{<c>}{<q> <Rn>!, <registers> // (Preferred syntax)
```

```
STMEA{<c>}{<q> <Rn>!, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = '00000000'::register_list;
let wback : boolean = TRUE;
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in [Using R15](#).

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	W	0					Rn	(0)	M													
opc							L				P					register_list															

Encoding

`STM{IA}{<c>}.W <Rn>{!}, <registers>` // (Preferred syntax, if <Rn>, '!' and <registers> can be represented in T1)

`STMEA{<c>}.W <Rn>{!}, <registers>` // (Alternate syntax, Empty Ascending stack, if <Rn>, '!' and <registers> can be represented in T1)

`STM{IA}{<c>}{<q>} <Rn>{!}, <registers>` // (Preferred syntax)

`STMEA{<c>}{<q>} <Rn>{!}, <registers>` // (Alternate syntax, Empty Ascending stack)

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in [Using R15](#).

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

IA	Is an optional suffix for the Increment After form.
<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	For the "A1" variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. For the "T1" variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. For the "T2" variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address) = ARBITRARY : bits(32); // Only possible for encodings T1 and A1
            else
                MemS{32}(address) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then // Only possible for encoding A1
        MemS{32}(address) = PCStoreValue();
    end;
    if wback then R(n) = R(n) + 4*BitCount(registers); end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

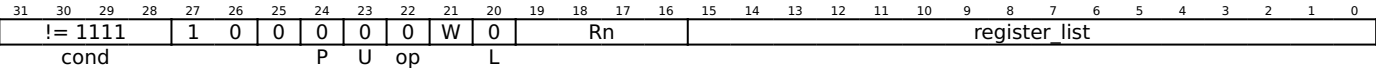
STMDA, STMED

Store Multiple Decrement After (Empty Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#). For details of related system instructions see [STM \(User registers\)](#).

A1



Encoding

```
STMDA{<c>}{<q>}<Rn>{!}, <registers> // (Preferred syntax)

STMED{<c>}{<q>}<Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = (R(n) - 4*BitCount(registers)) + 4;
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address) = ARBITRARY : bits(32);
            else
                MemS{32}(address) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        MemS{32}(address) = PCStoreValue();
    end;
    if wback then R(n) = R(n) - 4*BitCount(registers); end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STMDB, STMFD

Store Multiple Decrement Before (Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#). For details of related system instructions see [STM \(User registers\)](#).

This instruction is used by the alias [PUSH \(multiple registers\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	0	0	1	0	0	W	0	Rn				register list															
cond				P				U	op	L																					

Encoding

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an *STM* with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	W	0	Rn				(0)	M	register_list													
opc							L								P																

Encoding

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an *STM* with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The store instruction executes but the value stored for the base register is UNKNOWN.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	For the "A1" variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. For the "T1" variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Alias Conditions

Alias	Of variant	Is preferred when
PUSH (multiple registers)	A1	<code>W == '1' && Rn == '1101' && BitCount(register_list) > 1</code>
PUSH (multiple registers)	T1	<code>W == '1' && Rn == '1101' && BitCount(M :: register_list) > 1</code>
PUSH (multiple registers)	T1	<code>BitCount(M :: register_list) > 1</code>

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) - 4*BitCount(registers);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address) = ARBITRARY : bits(32); // Only possible for encoding A1
            else
                MemS{32}(address) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then // Only possible for encoding A1
        MemS{32}(address) = PCStoreValue();
    end;
    if wback then R(n) = R(n) - 4*BitCount(registers); end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

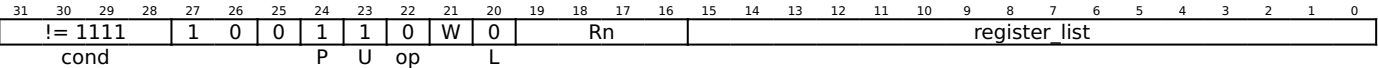
STMIB, STMFA

Store Multiple Increment Before (Full Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see [FEAT_LSMAOC](#). For details of related system instructions see [STM \(User registers\)](#).

A1



Encoding

```
STMIB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) + 4;
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address) = ARBITRARY : bits(32);
            else
                MemS{32}(address) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        MemS{32}(address) = PCStoreValue();
    end;
    if wback then R(n) = R(n) + 4*BitCount(registers); end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

This instruction is used by the alias [PUSH \(single register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) , [T3](#) and [T4](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	P	U	0	W	0	Rn				Rt				imm12											
cond				op2				op1																							

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn> {, # {+/-} <imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn>], # {+/-} <imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STR{<c>}{<q>} <Rt>, [<Rn>, # {+/-} <imm>]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Rn			Rt		
B			L												

Encoding

STR{<c>}{<q>} <Rt>, [<Rn> {, # {+} <imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'00');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rt			imm8							
L															

Encoding

STR{<c>}{<q>}<Rt>, [SP{, #{+}<imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = 13;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	1	0	0	!= 1111			Rt			imm12													
size								L								Rn															

Encoding

STR{<c>}{<q>}<Rt>, [<Rn> {, #{+}<imm>}]

STR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **t == 15**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	0	0	!= 1111			Rt			1	P	U	W	imm8								
size								L								Rn															

Encoding for the Offset variant

Applies when (P == 1 && U == 0 && W == 0)

STR{<c>}{<q>}<Rt>, [<Rn> {, #-<imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

STR{<c>}{<q>}<Rt>, [<Rn>], #{+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STR{<c>}{<q> <Rt>, [<Rn>, #{+/-}<imm>]!

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then See("STRT"); end;
if Rn == '1111' || (P == '0' && W == '0') then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated. For the "T1", "T2", "T3", "T4 Offset", "T4 Post-indexed", and "T4 Pre-indexed" variants: is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For the "T1", "T3", "T4 Offset", "T4 Post-indexed", and "T4 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field. For the "T1" variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4. For the "T2" variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4. For the "T3" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T4 Offset", "T4 Post-indexed", and "T4 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.						
+	Specifies the offset is added to the base register.						

Alias Conditions

Alias	Of variant	Is preferred when
PUSH (single register)	A1 Pre-indexed	<code>P == '1' && U == '0' && W == '1' && Rn == '1101' && imm12 == '00000000100'</code>
PUSH (single register)	T4 Pre-indexed	<code>Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100'</code>

Operation

```
if CurrentInstrSet\(\) == InstrSet A32 then
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{32}(address) = if t == 15 then PCStoreValue() else R(t);
    if wback then R(n) = offset_addr; end;
  end;
else
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{32}(address) = R(t);
    if wback then R(n) = offset_addr; end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	P	U	0	W	0	Rn				Rt				imm5				stype		0	Rm				
cond				op2				op1																							

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
if m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rm			Rn			Rt		
L				B		H									

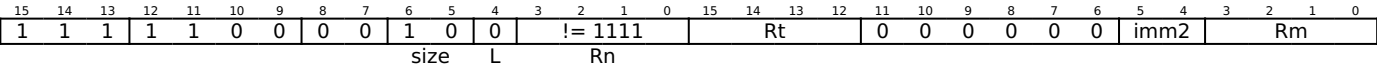
Encoding

STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTType = SRTType_LSL;
let shift_n : integer = 0;
```

T2



Encoding

```
STR{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `t == 15`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated. For the "T1" and "T2" variants: is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For the "T1" and "T2" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						
<shift>	The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register .						
+	Specifies the index register is added to the base register.						
<imm>	If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.						

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let offset : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
  let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
  let address : bits(32) = if index then offset_addr else R(n);
  var data : bits(32);
  if t == 15 then // Only possible for encoding A1
    data = PCStoreValue();
  else
    data = R(t);
  end;
  MemU{32}(address) = data;
  if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	P	U	1	W	0	Rn				Rt				imm12											
cond				op2				op1																							

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STRB{<c>}{<q>}<Rt>, [<Rn> {, # {+/-}<imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STRB{<c>}{<q>}<Rt>, [<Rn>], # {+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STRB{<c>}{<q>}<Rt>, [<Rn>, # {+/-}<imm>]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `t == 15`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
- If `wback && n == t`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The store instruction executes but the value stored is UNKNOWN.
- If `wback && n == 15`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The instruction executes without writeback of the base address.
 - The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	imm5					Rn			Rt		
B				L											

Encoding


```
STRB{<c>}{<q>}<Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	0	0	1	0	0	0	!	1	1	1	Rt				imm12															
								size				L				Rn																			

Encoding

```
STRB{<c>}{<q>}<Rt>, [<Rn> {, #{+}<imm>}]
```

```
STRB{<c>}.W<Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	0	0	0	0	0	0	!	=	1111		Rt					1	P	U	W	imm8										
												size				L				Rn															

Encoding for the Offset variant

Applies when (P == 1 && U == 0 && W == 0)

```
STRB{<c>}{<q>}<Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
STRB{<c>}{<q>}<Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
STRB{<c>}{<q>}<Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then See("STRBT"); end;
if Rn == '1111' || (P == '0' && W == '0') then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For the "T1", "T2", "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field. For the "T1" variant: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field. For the "T2" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.						
+	Specifies the offset is added to the base register.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{8}(address) = R(t)[7:0];
    if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	P	U	1	W	0	Rn				Rt				imm5				stype	0	Rm					
cond				op2				op1																							

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	Rm			Rn			Rt		
			L			B			H						

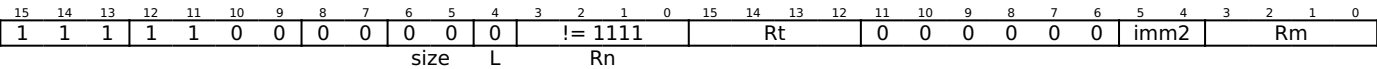
Encoding

STRB{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding

```
STRB{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

STRB{<c>}.W<Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-Ā removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

- If `t == 15`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as NOP.
 - The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For the "T1" and "T2" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						
<shift>	The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register .						
+	Specifies the index register is added to the base register.						
<imm>	If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{8}(address) = R(t)[7:0];
    if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRBT

Store Register Byte Unprivileged stores a byte from a register to memory. For information about memory accesses see [Memory accesses](#). The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	0	U	1	1	0	Rn				Rt				imm12											
cond				P				op2				W				op1															

Encoding

STRBT{<c>}{<q>}<Rt>, [<Rn>] {, #(<+/-><imm>)}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	U	1	1	0	Rn				Rt				imm5					stype	0	Rm				
cond				P				op2				W				op1															

Encoding

STRBT{<c>}{<q> <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **t == 15**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If **n == t**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If **n == 15**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	!	1111																	
size L												Rn				Rt				1 1 1 0				imm8							

Encoding

STRBT{<c>}{<q> <Rt>, [<Rn> {, #(+)<imm>}]

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{(imm8)};
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **t == 15**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Rt>	For the "A1" variant: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated. For the "A2" and "T1" variants: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn>	Is the general-purpose base register, encoded in the "Rn" field.												
+/-	For the "A1" variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table> <tr> <th>U</th><th>+/-</th></tr> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </table> For the "A2" variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table> <tr> <th>U</th><th>+/-</th></tr> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </table>	U	+/-	0	-	1	+	U	+/-	0	-	1	+
U	+/-												
0	-												
1	+												
U	+/-												
0	-												
1	+												
<imm>	For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field. For the "T1" variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.												
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.												
<shift>	The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see <i>Shifts applied to a register</i> .												
+	Specifies the offset is added to the base register.												

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end; // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                           else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    MemU_unpriv{8}(address) = R(t)[7:0];
    if postindex then R(n) = offset_addr; end;
end;

```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as STRB (immediate).

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	0	Rn				Rt				imm4H				1	1	1	1	imm4L			
cond												o1																op2			

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, # {+/-} <imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], # {+/-} <imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, # {+/-} <imm>]!

Decode for all variants of this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t || n == t2) then UnpredictableProcedure(); end;
if t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

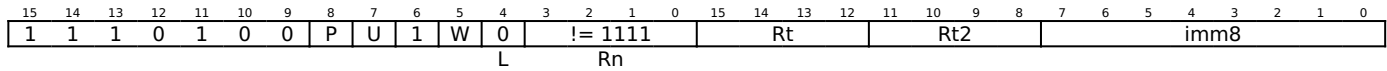
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `t<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when `Rt == '1111'`.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

T1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
STRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn> {, #<+/-><imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
STRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>], #<+/-><imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
STRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn>, #<+/-><imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '0' then See("Related encodings"); end;
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if wback && (n == t || n == t2) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || t == 15 || t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Load/store dual](#), [load/store exclusive](#), [table branch](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rt>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14. For the "T1 Offset", "T1 Post-indexed", and "T1 Pre-indexed" variants: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the second general-purpose register to be transferred. This register must be <R(t+1)>. For the "T1 Offset", "T1 Post-indexed", and "T1 Pre-indexed" variants: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For the "T1 Offset", "T1 Post-indexed", and "T1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table> <tr> <th>U</th><th>+/-</th></tr> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1 Offset", "T1 Post-indexed", and "T1 Pre-indexed" variants: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.						

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        var data : bits(64);
        if BigEndian(AccessType_GPR) then
            data[63:32] = R(t);
            data[31:0] = R(t2);
        else
            data[31:0] = R(t);
            data[63:32] = R(t2);
        end;
        MemA{64}(address) = data;
    else
        MemA{32}(address) = R(t);
        MemA{32}(address+4) = R(t2);
    end;
    if wback then R(n) = offset_addr; end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

STRD (register)

Store Register Dual (register) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	0	W	0	Rn				Rt				(0) (0) (0) (0) 1				1	1	1	Rm				
cond				o1								op2																			

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!

Decode for all variants of this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if t2 == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t || n == t2) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `t<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when `Rt == '1111'`.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `P = '1'; W = '0'`.
- The instruction executes with the additional decode: `P = '1'; W = '1'`.

- The instruction executes with the additional decode: P = '0'; W = '0'.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.
- <Rt2> Is the second general-purpose register to be transferred. This register must be <R(t+1)>.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.
- +/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

- <Rm> Is the general-purpose index register, encoded in the "Rm" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + R(m)) else (R(n) - R(m));
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        var data : bits(64);
        if BigEndian(AccessType GPR) then
            data[63:32] = R(t);
            data[31:0] = R(t2);
        else
            data[31:0] = R(t);
            data[63:32] = R(t2);
        end;
        MemA{64}(address) = data;
    else
        MemA{32}(address) = R(t);
        MemA{32}(address+4) = R(t2);
    end;
    if wback then R(n) = offset_addr; end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, stores a word from a register to the calculated address if the PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	0	0	0	Rn				Rd				(1)	(1)	1	1	1	0	0	1	Rt			
cond				size L								ex ord																			

Encoding

STREX{<c>}{<q>}<Rd>, <Rt>, [<Rn> {, {#}<imm>}]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = Zeros{}; // Zero offset
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	0	Rn				Rt				Rd				imm8							
op0												L																			

Encoding

STREX{<c>}{<q>}<Rd>, <Rt>, [<Rn> {, #<imm>}]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<imm>	For the "A1" variant: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted. For the "T1" variant: the immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is `IMPLEMENTATION DEFINED` whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is `IMPLEMENTATION DEFINED` whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n) + imm32;
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemA{32}(address) = R(t);
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STREXB

Store Register Exclusive Byte derives an address from a base register value, stores a byte from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	0	0	Rn				Rd				(1)	(1)	1	1	1	0	0	1	Rt			
cond				size								L								ex				ord							

Encoding

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	0	1	0	0	Rd			
				op0								L								Rt2				sz							

Encoding

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemA{8}(address) = R(t)[7:0];
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STREXD

Store Register Exclusive Doubleword derives an address from a base register value, stores a 64-bit doubleword from two registers to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	1	0	1	0	Rn				Rd				(1)	(1)	1	1	1	0	0	1	Rt			
cond				size								L								ex				ord							

Encoding

STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
if d == 15 || Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `Rt<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If `Rt == '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in [Using R15](#).

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				Rt2				0	1	1	1	Rd			
op0												L												sz							

Encoding

STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory. <Rd> must not be the same as <Rn>, <Rt>, or <Rt2>.
<Rt>	For the "A1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14. For the "T1" variant: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For the "A1" variant: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>. For the "T1" variant: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non doubleword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);

    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    let value : bits(64) = if BigEndian(AccessType_GPR) then R(t)::R(t2) else R(t2)::R(t);

    if AArch32.ExclusiveMonitorsPass(address,8) then
        MemA{64}(address) = value;
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

STREXH

Store Register Exclusive Halfword derives an address from a base register value, stores a halfword from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	1	1	0	Rn				Rd				(1)	(1)	1	1	1	0	0	1	Rt			
cond								size				L								ex				ord							

Encoding

STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	0	1	0	1	Rd			
				op0				L								Rt2				sz											

Encoding

STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemA{16}(address) = R(t)[15:0];
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	0	Rn				Rt				imm4H				1	0	1	1	imm4L			
cond											o1					op2															

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STRH{<c>}{<q>}<Rt>, [<Rn> {, # {+/-}<imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STRH{<c>}{<q>}<Rt>, [<Rn>], # {+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STRH{<c>}{<q>}<Rt>, [<Rn>, # {+/-}<imm>]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	imm5						Rn			Rt	
L															

Encoding

```
STRH{<c>}{<q>}<Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'0');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	0	0	0	1	0	1	0	!= 1111					Rt					imm12											
size											L		Rn																				

Encoding

```
STRH{<c>}{<q>}<Rt>, [<Rn> {, #{+}<imm>}]
STRH{<c>}.W<Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	0	!= 1111				Rt				1	P	U	W	imm8							
size										L		Rn																			

Encoding for the Offset variant

Applies when (P == 1 && U == 0 && W == 0)

```
STRH{<c>}{<q>}<Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
STRH{<c>}{<q>}<Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
STRH{<c>}{<q>}<Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then See("STRHT"); end;
if Rn == '1111' || (P == '0' && W == '0') then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For the "A1 Offset" variant: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For the "A1 Post-indexed", "A1 Pre-indexed", "T1", "T2", "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1" variant: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2. For the "T2" variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the "T3 Offset", "T3 Post-indexed", and "T3 Pre-indexed" variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.						
+	Specifies the offset is added to the base register.						

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{16}(address) = R(t)[15:0];
    if wback then R(n) = offset_addr; end;
  end;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{16}(address) = R(t)[15:0];
    if wback then R(n) = offset_addr; end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see [Memory accesses](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	0	W	0	Rn				Rt				(0) (0) (0) (0)				1	0	1	1	Rm			
cond											o1					op2															

Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STRH{<c>}{<q>}<Rt>, [<Rn>, {+/-}<Rm>]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STRH{<c>}{<q>}<Rt>, [<Rn>], {+/-}<Rm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STRH{<c>}{<q>}<Rt>, [<Rn>, {+/-}<Rm>]!

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	Rm			Rn			Rt		
L				B		H									

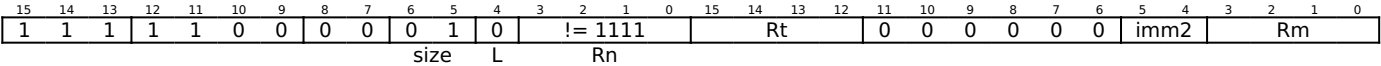
Encoding

STRH{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding

```
STRH{<c>}{<q>}<Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

STRH{<c>}.W<Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-Ā removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For the "A1 Offset", "A1 Post-indexed", and "A1 Pre-indexed" variants: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For the "T1" and "T2" variants: is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						
+	Specifies the index register is added to the base register.						
<imm>	If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.						

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let offset : bits(32) = Shift{(R(m), shift_t, shift_n, PSTATE.C);
  let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
  let address : bits(32) = if index then offset_addr else R(n);
  MemU{16}(address) = R(t)[15:0];
  if wback then R(n) = offset_addr; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRHT

Store Register Halfword Unprivileged stores a halfword from a register to memory. For information about memory accesses see [Memory accesses](#). The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	U	1	1	0	Rn				Rt				imm4H				1	0	1	1	imm4L			
cond				P				W				o1												op2							

Encoding

STRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	U	0	1	0	Rn				Rt				(0)	(0)	(0)	(0)	1	0	1	1	Rm			
cond				P				W				o1												op2							

Encoding

STRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if `bit[24] == 1` and `bit[21] == 0`. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	0	!= 1111				Rt				1	1	1	0	imm8							
size												L	Rn																		

Encoding

STRHT{<c>}{<q>}<Rt>, [<Rn> {, #(+)<imm>}]

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend()(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
+/-	For the "A1" variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

For the "A2" variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm>	For the "A1" variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field. For the "T1" variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.
+	Specifies the offset is added to the base register.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;           // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    MemU_unpriv{16}(address) = R(t)[15:0];
    if postindex then R(n) = offset_addr; end;
end;

```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as STRH (immediate).

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRT

Store Register Unprivileged stores a word from a register to memory. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	0	U	0	1	0	Rn				Rt				imm12											
cond				P				op2				W				op1															

Encoding

STRT{<c>}{<q>} <Rt>, [<Rn>] {, # {+/-} <imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
if n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	U	0	1	0	Rn				Rt				imm5				stype		0	Rm				
cond				P				op2				W				op1															

Encoding

STRT{<c>}{<q>} <Rt>, [<Rn>], {+/-} <Rm>{, <shift>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if `bit[24] == 1` and `bit[21] == 0`. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	1	0	0	0	0	1	0	0	!= 1111				Rt				1				1	1	0	imm8							
size L												Rn																						

Encoding

STRT{<c>}{<q>}<Rt>, [<Rn> {, #(<+>)<imm>}]

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .												
<q>	See Standard assembler syntax fields .												
<Rt>	For the "A1" and "A2" variants: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated. For the "T1" variant: is the general-purpose register to be transferred, encoded in the "Rt" field.												
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.												
+/-	For the "A1" variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table> For the "A2" variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+	U	+/-	0	-	1	+
U	+/-												
0	-												
1	+												
U	+/-												
0	-												
1	+												
<imm>	For the "A1" variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.												

For the "T1" variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end; // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                           else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    var data : bits(32);
    if t == 15 then // Only possible for encodings A1 and A2
        data = PCStoreValue();
    else
        data = R(t);
    end;
    MemU_unpriv{32}(address) = data;
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as STR (immediate).

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (immediate, from PC)

Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This is an alias of [ADR](#). This means:

- The encodings in this description are named to match the encodings of [ADR](#).
- The description of [ADR](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A2](#)) and T32 ([T2](#)) .

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	1	0	0	1	0	0	1	1	1	1	Rd					imm12											
cond				opc								S	Rn																			

Encoding for the A2 variant

SUB{<c>} {<q>} <Rd>, PC, #<const>

is equivalent to

[ADR](#){<c>} {<q>} <Rd>, <label>

and is the preferred disassembly when `imm12 == '000000000000'`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	i	1	0	1	0	1	0	1	1	1	1	0	imm3				Rd				imm8										
o1																o2				Rn															

Encoding for the T2 variant

SUB{<c>} {<q>} <Rd>, PC, #<imm12>

is equivalent to

[ADR](#){<c>} {<q>} <Rd>, <label>

and is the preferred disassembly when `i :: imm3 :: imm8 == '000000000000'`.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A2" variant: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC . For the "T2" variant: is the general-purpose destination register, encoded in the "Rd" field.
<const>	An immediate value. See Modified immediate constants in A32 instructions for the range of values.
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

Operation

The description of [ADR](#) gives the operational pseudocode for this instruction.

SUB, SUBS (immediate)

Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SUBS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
 - The instruction is UNDEFINED in Hyp mode, except for encoding T5 with <imm8> set to zero, which is the encoding for the ERET instruction, see *ERET*.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) , [T3](#) , [T4](#) and [T5](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	0	1	0	S	Rn				Rd				imm12											
cond				opc																											

Encoding for the SUB variant

Applies when (S == 0 && Rn != 11x1)

SUB{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the SUBS variant

Applies when (S == 1 && Rn != 1101)

SUBS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
if Rn == '1111' && S == '0' then See("ADR"); end;
if Rn == '1101' then See("SUB (SP minus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	imm3			Rn			Rd		
S															

Encoding

SUB<c>{<q>} <Rd>, <Rn>, #<imm3> // (InITBlock())

SUBS{<q>} <Rd>, <Rn>, #<imm3> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm3);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	Rdn			imm8							
op															

Encoding

```

SUB<c>{<q>} <Rdn>, #<imm8> // (InITBlock() && UInt(Rdn) < 8 && UInt(imm8) < 8)

SUB<c>{<q>} {<Rdn>, }<Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)

SUBS{<q>} <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)

SUBS{<q>} {<Rdn>, }<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)

```

Decode for this encoding

```

let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm8);

```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	1	1	0	1	S	!= 1101				0	imm3				Rd				imm8							
op1												Rn																				

Encoding for the SUB variant

Applies when (S == 0)

```

SUB{<c>}{<q>} {<Rd>, }<Rn>, #<const>

SUB<c>.W {<Rd>, }<Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2)

```

Encoding for the SUBS variant

Applies when (S == 1 && Rd != 1111)

```

SUBS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

SUBS.W {<Rd>, }<Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2)

```

Decode for all variants of this encoding

```

if Rd == '1111' && S == '1' then See("CMP (immediate)"); end;
if Rn == '1101' then See("SUB (SP minus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;

```

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	i	1	0	1	0	1	0	!= 11x1				0	imm3				Rd				imm8								
o1						o2		Rn																									

Encoding

```

SUB{<c>}{<q>} {<Rd>, }<Rn>, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)

SUBW{<c>}{<q>} {<Rd>, }<Rn>, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

```

Decode for this encoding

```

if Rn == '1111' then See("ADR"); end;
if Rn == '1101' then See("SUB (SP minus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;

```

T5

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	1	Rn				1	0	(0)	0	(1)	(1)	(1)	(1)	imm8							

Encoding

Applies when (!(Rn == 1110 && imm8 == 00000000))

SUBS{<c>}{<q>} PC, LR, #<imm8>

Decode for this encoding

```
if Rn == '1110' && IsZero(imm8) then See("ERET"); end;
let d : integer = 15;
let n : integer = UInt(Rn);
let setflags : boolean = TRUE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
if n != 14 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [SUBS PC, LR and related instructions \(A32\)](#) and [SUBS PC, LR and related instructions \(T32\)](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	For the "A1 SUB" and "A1 SUBS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. If the PC is used: - For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. - For the SUBS variant, the instruction performs an exception return, that restores <i>PSTATE</i> from SPSR_<current_mode>. Arm deprecates use of this instruction unless <Rn> is the LR.
<Rn>	For the "T1", "T3 SUB", "T3 SUBS", and "T4" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. For the "A1 SUB", "A1 SUBS", and "T4" variants: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB (SP minus immediate). If the PC is used, see ADR. For the "T1" variant: is the general-purpose source register, encoded in the "Rn" field. For the "T3 SUB" and "T3 SUBS" variants: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB (SP minus immediate).
<const>	For the "A1 SUB" and "A1 SUBS" variants: an immediate value. See Modified immediate constants in A32 instructions for the range of values. For the "T3 SUB" and "T3 SUBS" variants: an immediate value. See Modified immediate constants in T32 instructions for the range of values.
<imm3>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.
<Rdn>	Is the general-purpose source and destination register, encoded in the "Rdn" field.
<imm8>	For the "T2" variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. For the "T5" variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. If <Rn> is the LR, and zero is used, see ERET.
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
In the T32 instruction set, MOVS{<c>}{<q>} PC, LR is a pseudo-instruction for SUBS{<c>}{<q>} PC, LR, #0.	

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(imm32), '1');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SUB, SUBS (register)

Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	1	0	S	!= 1101				Rd				imm5				stype		0	Rm				
cond				opc				Rn																							

Encoding for the SUB, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

SUB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SUB, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

SUB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the SUBS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

SUBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SUBS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

SUBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
if Rn == '1101' then See("SUB (SP minus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	Rm			Rn			Rd		
S															

Encoding

SUB{<c>}{<q>} <Rd>, <Rn>, <Rm> // (InITBlock())

SUBS{<q>} {<Rd>, }<Rn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	0	1	S	!= 1101				(0)	imm3			Rd			imm2			stype		Rm			
op1												Rn																			

Encoding for the SUB, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

SUB{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SUB, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SUB{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

SUB<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the SUBS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

SUBS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SUBS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SUBS{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

SUBS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (register)"); end;
if Rn == '1101' then See("SUB (SP minus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> For the "A1 SUB, rotate right with extend", "A1 SUB, shift or rotate by value", "A1 SUBS, rotate right with extend", and "A1 SUBS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the SUBS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the "T1", "T2 SUB, rotate right with extend", "T2 SUB, shift or rotate by value", "T2 SUBS, rotate right with extend", and "T2 SUBS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For the "A1 SUB, rotate right with extend", "A1 SUB, shift or rotate by value", "A1 SUBS, rotate right with extend", and "A1 SUBS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. If the SP is used, see [SUB \(SP minus register\)](#).

For the "T1" variant: is the first general-purpose source register, encoded in the "Rn" field.

For the "T2 SUB, rotate right with extend", "T2 SUB, shift or rotate by value", "T2 SUBS, rotate right with extend", and "T2 SUBS, shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see [SUB \(SP minus register\)](#).

<Rm> For the "A1 SUB, rotate right with extend", "A1 SUB, shift or rotate by value", "A1 SUBS, rotate right with extend", and "A1 SUBS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For the "T1", "T2 SUB, rotate right with extend", "T2 SUB, shift or rotate by value", "T2 SUBS, rotate right with extend", and "T2 SUBS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the "A1 SUB, shift or rotate by value" and "A1 SUBS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For the "T2 SUB, shift or rotate by value" and "T2 SUBS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
  let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
  if d == 15 then // Can only occur for A32 encoding
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
    end;
  else
    R(d) = result;
    if setflags then
      PSTATE.[N,Z,C,V] = nzcv;
    end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

SUB, SUBS (register-shifted register)

Subtract (register-shifted register) subtracts a register-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	1	0	S	Rn				Rd				Rs				0	stype		1	Rm			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

SUBS{<c>}{<q>}{<Rd>}, <Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

SUB{<c>}{<q>}{<Rd>}, <Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”:										
<table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>		stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<Rs>	Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{R(m), shift_t, shift_n, PSTATE.C};
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SUB, SUBS (SP minus immediate)

Subtract from SP (immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	0	1	0	S	1	1	0	1	Rd				imm12											
cond								opc				Rn																			

Encoding for the SUB variant

Applies when (S == 0)

SUB{<c>}{<q>} {<Rd>, }SP, #<const>

Encoding for the SUBS variant

Applies when (S == 1)

SUBS{<c>}{<q>} {<Rd>, }SP, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	1	imm7						
								S							

Encoding

SUB{<c>}{<q>} {SP, }SP, #<imm7>

Decode for this encoding

```
let d : integer = 13;
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm7::'00');
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	1	1	0	1	S	1	1	0	1	0	imm3				Rd				imm8							
op1																Rn																

Encoding for the SUB variant

Applies when (S == 0)

SUB{<c>}{<q>} {<Rd>, }SP, #<const>

SUB{<c>}.W {<Rd>, }SP, #<const> // (<Rd>, <const> can be represented in T1)

Encoding for the SUBS variant

Applies when (S == 1 && Rd != 1111)

SUBS{<c>}{<q>}{<Rd>, }SP, #<const>

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (immediate)"); end;
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if d == 15 && !setflags then UnpredictableProcedure(); end;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	1	0	1	0	1	0	1	1	0	1	0	imm3				Rd				imm8							
								o1		o2						Rn																

Encoding

SUB{<c>}{<q>}{<Rd>, }SP, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)

SUBW{<c>}{<q>}{<Rd>, }SP, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> For the "A1 SUB" and "A1 SUBS" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the SUBS variant, the instruction performs an exception return, that restores **PSTATE** from SPSR_<current_mode>. Arm deprecates use of this instruction unless <Rn> is the LR.

For the "T2 SUB", "T2 SUBS", and "T3" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<const> For the "A1 SUB" and "A1 SUBS" variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the "T2 SUB" and "T2 SUBS" variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

SP Is the stack pointer.

<imm7> Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), NOT(imm32), '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

SUB, SUBS (SP minus register)

Subtract from SP (register) subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register. If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	1	0	S	1	1	0	1	Rd			imm5			stype		0	Rm						
cond				opc				Rn																							

Encoding for the SUB, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

```
SUB{<c>}{<q>} {<Rd>, }SP, <Rm> , RRX
```

Encoding for the SUB, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

```
SUB{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Encoding for the SUBS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

```
SUBS{<c>}{<q>} {<Rd>, }SP, <Rm> , RRX
```

Encoding for the SUBS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

```
SUBS{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	0	1	1	1	0	1	S	1	1	0	1	(0)	imm3			Rd			imm2			stype		Rm				
op1							Rn																									

Encoding for the SUB, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
SUB{<c>}{<q>} {<Rd>, }SP, <Rm> , RRX
```

Encoding for the SUB, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
SUB{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>} // (Preferred syntax)
```

```
SUB{<c>}.W {<Rd>, }SP, <Rm>
```

Encoding for the SUBS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

SUBS{<c>}{<q>}{<Rd>, }SP, <Rm>, RRR

Encoding for the SUBS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SUBS{<c>}{<q>}{<Rd>, }SP, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (register)"); end;
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	For the "A1 SUB, rotate right with extend", "A1 SUB, shift or rotate by value", "A1 SUBS, rotate right with extend", and "A1 SUBS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used: • For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. • For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.										
<Rm>	For the "T1 SUB, rotate right with extend", "T1 SUB, shift or rotate by value", "T1 SUBS, rotate right with extend", and "T1 SUBS, shift or rotate by value" variants: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. For the "A1 SUB, rotate right with extend", "A1 SUB, shift or rotate by value", "A1 SUBS, rotate right with extend", and "A1 SUBS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T1 SUB, rotate right with extend", "T1 SUB, shift or rotate by value", "T1 SUBS, rotate right with extend", and "T1 SUBS, shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”: <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the "A1 SUB, shift or rotate by value" and "A1 SUBS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32. For the "T1 SUB, shift or rotate by value" and "T1 SUBS, shift or rotate by value" variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), NOT(shifted), '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.\[N,Z,C,V\] = nzcvc;
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SVC

Supervisor Call causes a Supervisor Call exception. For more information, see [Supervisor Call \(SVC\) exception](#).

Note

SVC was previously called SWI, Software Interrupt, and this name is still found in some documentation.

Software can use this instruction as a call to an operating system to provide a service.

In the following cases, the Supervisor Call exception generated by the SVC instruction is taken to Hyp mode:

- If the SVC is executed in Hyp mode.
- If HCR.TGE is set to 1, and the SVC is executed in Non-secure User mode. For more information, see [Supervisor Call exception, when HCR.TGE is set to 1](#)

In these cases, the [HSR, Hyp Syndrome Register](#) identifies that the exception entry was caused by a Supervisor Call exception, EC value 0x11, see [Use of the HSR](#). The immediate field in the [HSR](#):

- If the SVC is unconditional:
 - For the T32 instruction, is the zero-extended value of the imm8 field.
 - For the A32 instruction, is the least-significant 16 bits the imm24 field.
- If the SVC is conditional, is UNKNOWN.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	1	imm24																							
cond																															

Encoding

SVC{<c>}{<q>}{#}<imm>

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{(imm24);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							
S															

Encoding

SVC{<c>}{<q>}{#}<imm>

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{(imm8);
```

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <imm> For the "A1" variant: is a 24-bit unsigned immediate, in the range 0 to 16777215, encoded in the "imm24" field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm24 in software, for example to determine the required service.
For the "T1" variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm8 in software, for example to determine the required service.

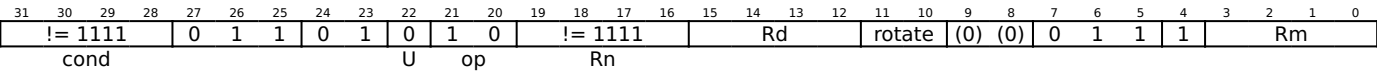
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_CheckForSVCTrap(imm32[15:0]);
    AArch32_CallSupervisor(imm32[15:0]);
end;
```


SXTAB

Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



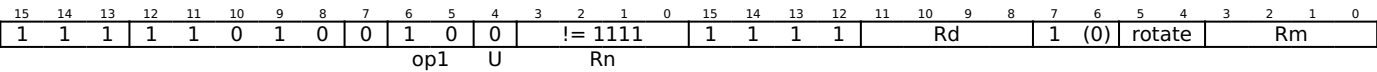
Encoding

```
SXTAB{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("SXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
SXTAB{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("SXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <amount> Is the rotate amount, encoded in "rotate":

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + SignExtend{32}(rotated[7:0]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SXTAB16

Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	0	0	!= 1111				Rd				rotate		(0)	(0)	0	1	1	1	Rm			
cond				U				op				Rn																			

Encoding

SXTAB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	0	!= 1111				1	1	1	1	Rd				1	(0)	rotate	Rm				
op1												U				Rn															

Encoding

SXTAB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<amount>	Is the rotate amount, encoded in “rotate”:										
<table><tr><th>rotate</th><th><amount></th></tr><tr><td>00</td><td>[absent]</td></tr><tr><td>01</td><td>8</td></tr><tr><td>10</td><td>16</td></tr><tr><td>11</td><td>24</td></tr></table>		rotate	<amount>	00	[absent]	01	8	10	16	11	24
rotate	<amount>										
00	[absent]										
01	8										
10	16										
11	24										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0] = R(n)[15:0] + SignExtend{16}(rotated[7:0]);
    R(d)[31:16] = R(n)[31:16] + SignExtend{16}(rotated[23:16]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

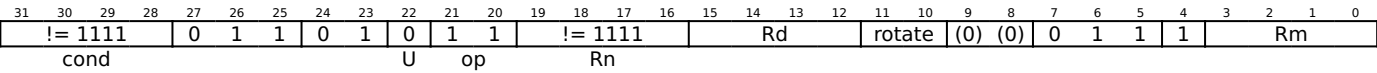
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTAH

Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



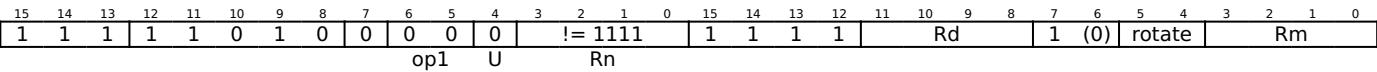
Encoding

SXTAH{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SXTAH{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <amount> Is the rotate amount, encoded in "rotate":

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + SignExtend{32}(rotated[15:0]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm					
cond				U				op				Rn																			

Encoding

SXTB{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	1	Rm			Rd		
U									B						

Encoding

SXTB{<c>}{<q>} {<Rd>, }<Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
1	1	1	1	1	0	1	0	0	1	0	0	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate	Rm										
op1																U		Rn																			

Encoding

SXTB{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

SXTB{<c>}.W {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = SignExtend{32}(rotated[7:0]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTB16

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	0	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			
cond				U				op				Rn																			

Encoding

SXTB16{<c>}{<q>}{<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
1	1	1	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate	Rm																				
op1																U																Rn															

Encoding

SXTB16{<c>}{<q>}{<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.
- <amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0] = SignExtend{16}(rotated[7:0]);
    R(d)[31:16] = SignExtend{16}(rotated[23:16]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	1	1	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			
cond				U				op				Rn																			

Encoding

SXTH{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	0	Rm			Rd		
				U				B							

Encoding

SXTH{<c>}{<q>} {<Rd>, }<Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate	Rm				
op1												U				Rn															

Encoding

SXTH{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

SXTH{<c>}.W {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = SignExtend{32}(rotated[15:0]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

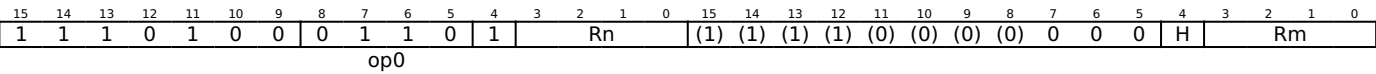
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBB, TBH

Table Branch Byte or Halfword causes a PC-relative forward branch using a table of single byte or halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value returned from the table.

T1



Encoding for the Byte variant

Applies when (H == 0)

```
TBB{<c>}{<q>} [<Rn>, <Rm>] // (Outside or last in IT block)
```

Encoding for the Halfword variant

Applies when (H == 1)

```
TBH{<c>}{<q>} [<Rn>, <Rm>, LSL #1] // (Outside or last in IT block)
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let is_tbh : boolean = (H == '1');
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rn> Is the general-purpose base register holding the address of the table of branch lengths, encoded in the "Rn" field. The PC can be used. If it is, the table immediately follows this instruction.
- <Rm> For the "Byte" variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.
For the "Halfword" variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

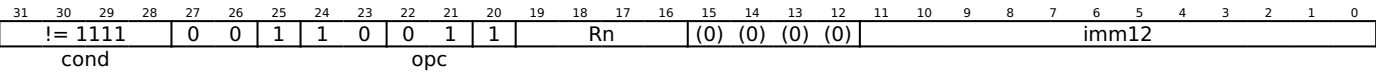
Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  var halfwords : integer;
  if is_tbh then
    halfwords = UInt(MemU{16}(R(n)+LSL(R(m), 1)));
  else
    halfwords = UInt(MemU{8}(R(n)+R(m)));
  end;
  BranchWritePC(PC32() + 2*halfwords, BranchType_INDIR);
end;
```

TEQ (immediate)

Test Equivalence (immediate) performs a bitwise exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



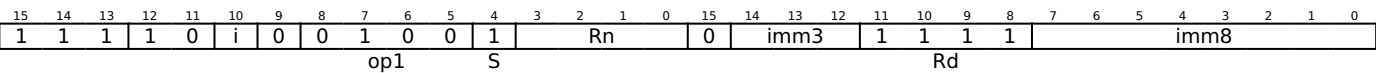
Encoding

TEQ{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1



Encoding

TEQ{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> For the "A1" variant: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For the "T1" variant: is the general-purpose source register, encoded in the "Rn" field.
- <const> For the "A1" variant: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.
For the "T1" variant: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                           else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) XOR imm32;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

TEQ (register)

Test Equivalence (register) performs a bitwise exclusive-OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0	0	0	1	0	0	1	1	Rn				(0)	(0)	(0)	(0)	imm5				stype		0	Rm								
cond									opc																										

Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

TEQ{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

Applies when (!(imm5 == 00000 && stype == 11))

TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTtype, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	0	1	0	1	0	0	1	Rn				(0)	imm3				1	1	1	1	imm2		stype		Rm			
op1												S	Rd																			

Encoding for the Rotate right with extend variant

Applies when (imm3 == 000 && imm2 == 00 && stype == 11)

TEQ{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

Applies when (!(imm3 == 000 && imm2 == 00 && stype == 11))

TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTtype, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> For the "A1 Rotate right with extend" and "A1 Shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For the "T1 Rotate right with extend" and "T1 Shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> For the "A1 Rotate right with extend" and "A1 Shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For the "T1 Rotate right with extend" and "T1 Shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the "A1 Shift or rotate by value" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For the "T1 Shift or rotate by value" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

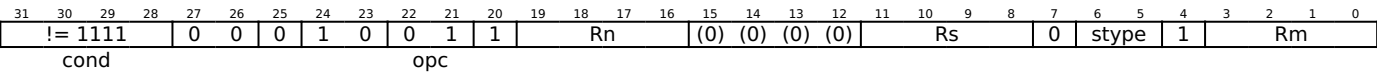
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TEQ (register-shifted register)

Test Equivalence (register-shifted register) performs a bitwise exclusive-OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1



Encoding

TEQ{<c>} {<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SType = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <type> Is the type of shift to be applied to the second source register, encoded in "stype":

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

TSB

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions, see *Trace Synchronization Barrier (TSB)*.
If *FEAT_TRF* is not implemented, this instruction executes as a NOP.
It has encodings from the following instruction sets: A32 (*Al*) and T32 (*Tl*) .

A1
(FEAT_TRF)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	0	1	0
cond				R				imm4				imm12																			

Encoding

TSB{<c>}{<q>} CSYNC

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then ExecuteAsNOP(); end;  
if cond != '1110' then UnpredictableProcedure(); end; // TSB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If *cond* != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1
(FEAT_TRF)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	0	1	0
																hint								option							

Encoding

TSB{<c>}{<q>} CSYNC

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then ExecuteAsNOP(); end;  
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If *InITBlock()*, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

Assembler Symbols

<c> See *Standard assembler syntax fields*.
<q> See *Standard assembler syntax fields*.

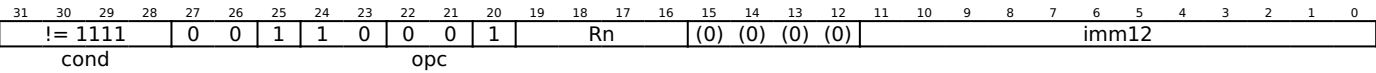
Operation

```
if ConditionPassed() then  
  EncodingSpecificOperations();  
  TraceSynchronizationBarrier();  
end;
```

TST (immediate)

Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



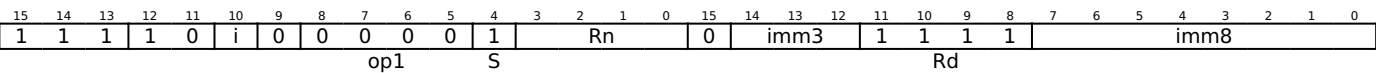
Encoding

TST{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1



Encoding

TST{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the `CONstrained UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> For the "A1" variant: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For the "T1" variant: is the general-purpose source register, encoded in the "Rn" field.
- <const> For the "A1" variant: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.
For the "T1" variant: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                           else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) AND imm32;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

TST (register)

Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111								0	0	0	1	0	0	0	1	Rn				(0)	(0)	(0)	(0)	imm5				stype	0	Rm			
cond								opc																									

Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when (!(imm5 == 00000 && stype == 11))

```
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	0	Rm			Rn		
op															

Encoding

```
TST{<c>}{<q>} <Rn>, <Rm>
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTType = SRTType_LSL;
let shift_n : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	0	0	1	Rn			(0)	imm3			1	1	1	1	imm2		stype		Rm				
op1											S	Rd																			

Encoding for the Rotate right with extend variant

Applies when (imm3 == 000 && imm2 == 00 && stype == 11)

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

Encoding for the Shift or rotate by value variant

Applies when (!(imm3 == 000 && imm2 == 00 && stype == 11))

```
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

```
TST{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rn>	For the "A1 Rotate right with extend" and "A1 Shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the "T1", "T2 Rotate right with extend", and "T2 Shift or rotate by value" variants: is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	For the "A1 Rotate right with extend" and "A1 Shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For the "T1", "T2 Rotate right with extend", and "T2 Shift or rotate by value" variants: is the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the type of shift to be applied to the second source register, encoded in "stype": <table><tr><th>stype</th><th><shift></th></tr><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></table>	stype	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
stype	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the "A1 Shift or rotate by value" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32. For the "T2 Shift or rotate by value" variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift\_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

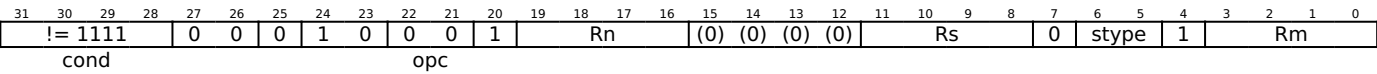
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

TST (register-shifted register)

Test (register-shifted register) performs a bitwise AND operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1



Encoding

TST{<c>} {<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SType = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <type> Is the type of shift to be applied to the second source register, encoded in "stype":

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	0	1	Rn				Rd				(1)	(1)	(1)	(1)	0	0	0	1	Rm			
cond				op1																B				op2							

Encoding

UADD16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	0	1	0	0	1	Rn				1	1	1	1	Rd				0	1	0	0	Rm							
op1												U																H				S			

Encoding

UADD16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[15:0]) + UInt(R(m)[15:0]);
    let sum2 : integer = UInt(R(n)[31:16]) + UInt(R(m)[31:16]);
    R(d)[15:0] = sum1[15:0];
    R(d)[31:16] = sum2[15:0];
    PSTATE.GE[1:0] = if sum1 >= 0x10000 then '11' else '00';
    PSTATE.GE[3:2] = if sum2 >= 0x10000 then '11' else '00';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

UADD8

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	0	1	Rn				Rd				(1)	(1)	(1)	(1)	1	0	0	1	Rm			
cond				op1												B				op2											

Encoding

UADD8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	0	Rn				1	1	1	1	Rd				0	1	0	0	Rm			
op1												U H S																			

Encoding

UADD8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[7:0]) + UInt(R(m)[7:0]);
    let sum2 : integer = UInt(R(n)[15:8]) + UInt(R(m)[15:8]);
    let sum3 : integer = UInt(R(n)[23:16]) + UInt(R(m)[23:16]);
    let sum4 : integer = UInt(R(n)[31:24]) + UInt(R(m)[31:24]);
    R(d)[7:0] = sum1[7:0];
    R(d)[15:8] = sum2[7:0];
    R(d)[23:16] = sum3[7:0];
    R(d)[31:24] = sum4[7:0];
    PSTATE.GE[0] = if sum1 >= 0x100 then '1' else '0';
    PSTATE.GE[1] = if sum2 >= 0x100 then '1' else '0';
    PSTATE.GE[2] = if sum3 >= 0x100 then '1' else '0';
    PSTATE.GE[3] = if sum4 >= 0x100 then '1' else '0';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UASX

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets *PSTATE*.GE according to the results. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	0	1	Rn				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rm			
cond				op1																B				op2							

Encoding

UASX{<c>}{<q>}{<Rd>, <Rn>, <Rm>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	0	Rn				1	1	1	1	Rd				0	1	0	0	Rm			
op1												U												H				S			

Encoding

UASX{<c>}{<q>}{<Rd>, <Rn>, <Rm>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = UInt(R(n)[15:0]) - UInt(R(m)[31:16]);
    let sum : integer = UInt(R(n)[31:16]) + UInt(R(m)[15:0]);
    R(d)[15:0] = diff[15:0];
    R(d)[31:16] = sum[15:0];
    PSTATE.GE[1:0] = if diff >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if sum >= 0x10000 then '11' else '00';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	1	1	widthm1				Rd				lsb				1 0		1	Rn					
cond										U																					

Encoding

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(lsb);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	1	1	0	0	Rn				0	imm3				Rd				imm2		(0)	widthm1			
op1																															

Encoding

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(imm3::imm2);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the "A1" variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.

For the "T1" variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.

<width> Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d) = ZeroExtend{32}(R(n)[msbit:lsbit]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDF

Permanently Undefined generates an Undefined Instruction exception.
The encodings for UDF used in this section are defined as permanently UNDEFINED. However:

- With the T32 instruction set, Arm deprecates using the UDF instruction in an IT block.
- In the A32 instruction set, UDF is not conditional.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	1	1	1	1	1	imm12												1	1	1	1	imm4			
cond																															

Encoding

UDF{<c>}{<q>}{#}<imm>

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{(imm12::imm4)};
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	0	imm8							
S															

Encoding

UDF{<c>}{<q>}{#}<imm>

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{(imm8)};
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	1	imm4				1	0	1	0	imm12											
o1												o2																			

Encoding

UDF{<c>}{<q>}{#}<imm>

UDF{<c>}.W {#}<imm> // (<imm> can be represented in T1)

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{(imm4::imm12)};
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

Assembler Symbols

- <c>
- For the "A1" variant: see *Standard assembler syntax fields*. <c> must be AL or omitted.
For the "T1" and "T2" variants: see *Standard assembler syntax fields*. Arm deprecates using any <c> value other than AL.
- <q>
- See *Standard assembler syntax fields*.
- <imm>
- For the "A1" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. The PE ignores the value of this constant.
For the "T1" variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant.
For the "T2" variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Undefined();
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.
See [Divide instructions](#) for more information about this instruction.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	0	1	1	Rd				(1)	(1)	(1)	(1)	Rm				0	0	0	1	Rn			
cond				op1								Ra								op2											

Encoding

UDIV{<c>}{<q> } {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **Ra != '1111'**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	0	1	1	Rn				(1)	(1)	(1)	(1)	Rd				1	1	1	1	Rm			
								op1								Ra								op2							

Encoding

UDIV{<c>}{<q> } {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a != 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **Ra != '1111'**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let dividend : integer = UInt(R(n));
    let divisor : integer = UInt(R(m));
    var result : integer;
    if divisor == 0 then
        result = 0;
    else
        result = dividend DIVRM divisor;
    end;
    R(d) = result[31:0];
end;
```

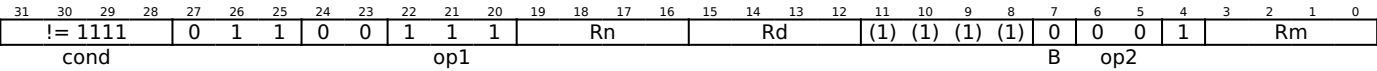
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



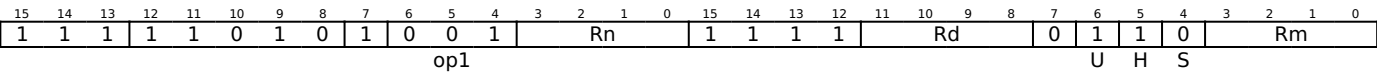
Encoding

```
UHADD16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
UHADD16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[15:0]) + UInt(R(m)[15:0]);
    let sum2 : integer = UInt(R(n)[31:16]) + UInt(R(m)[31:16]);
    R(d)[15:0] = sum1[16:1];
    R(d)[31:16] = sum2[16:1];
end;
```

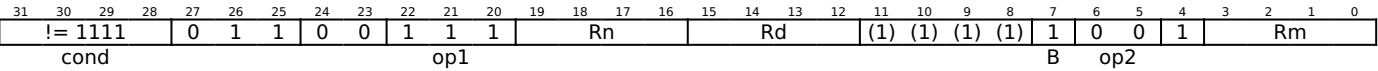
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UHADD8

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



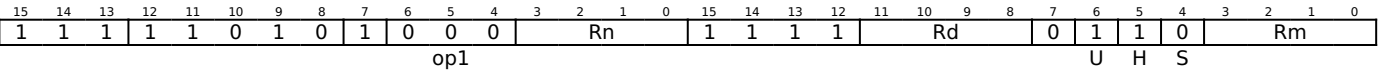
Encoding

UHADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UHADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[7:0]) + UInt(R(m)[7:0]);
    let sum2 : integer = UInt(R(n)[15:8]) + UInt(R(m)[15:8]);
    let sum3 : integer = UInt(R(n)[23:16]) + UInt(R(m)[23:16]);
    let sum4 : integer = UInt(R(n)[31:24]) + UInt(R(m)[31:24]);
    R(d)[7:0] = sum1[8:1];
    R(d)[15:8] = sum2[8:1];
    R(d)[23:16] = sum3[8:1];
    R(d)[31:24] = sum4[8:1];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UHASX

Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	1	1	Rn				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rm			
cond				op1																B				op2							

Encoding

UHASX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	0	Rn				1	1	1	1	Rd				0	1	1	0	Rm			
op1												U												H				S			

Encoding

UHASX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = UInt(R(n)[15:0]) - UInt(R(m)[31:16]);
    let sum : integer = UInt(R(n)[31:16]) + UInt(R(m)[15:0]);
    R(d)[15:0] = diff[16:1];
    R(d)[31:16] = sum[16:1];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UHSAX

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	1	1	Rn			Rd			(1) (1) (1) (1)				0	1	0	1	Rm					
cond				op1								B				op2															

Encoding

UHSAX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	1	0	Rn			1	1	1	1	Rd			0	1	1	0	Rm					
op1												U												H				S			

Encoding

UHSAX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = UInt(R(n)[15:0]) + UInt(R(m)[31:16]);
    let diff : integer = UInt(R(n)[31:16]) - UInt(R(m)[15:0]);
    R(d)[15:0] = sum[16:1];
    R(d)[31:16] = diff[16:1];
end;
```

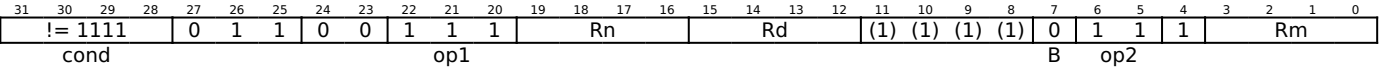
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UHSUB16

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



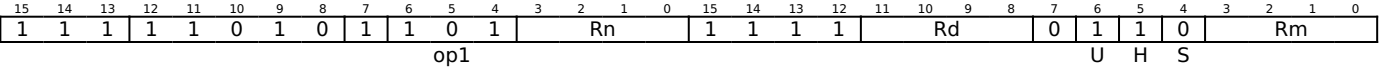
Encoding

```
UHSUB16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
UHSUB16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[15:0]) - UInt(R(m)[15:0]);
    let diff2 : integer = UInt(R(n)[31:16]) - UInt(R(m)[31:16]);
    R(d)[15:0] = diff1[16:1];
    R(d)[31:16] = diff2[16:1];
end;
```

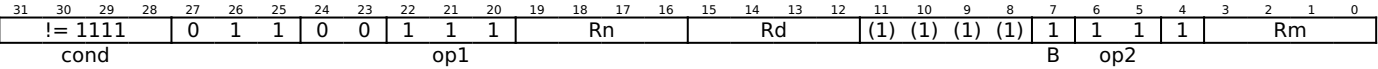
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UHSUB8

Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



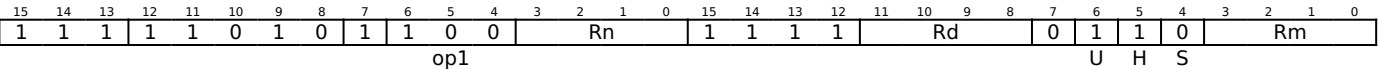
Encoding

UHSUB8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UHSUB8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[7:0]) - UInt(R(m)[7:0]);
    let diff2 : integer = UInt(R(n)[15:8]) - UInt(R(m)[15:8]);
    let diff3 : integer = UInt(R(n)[23:16]) - UInt(R(m)[23:16]);
    let diff4 : integer = UInt(R(n)[31:24]) - UInt(R(m)[31:24]);
    R(d)[7:0] = diff1[8:1];
    R(d)[15:8] = diff2[8:1];
    R(d)[23:16] = diff3[8:1];
    R(d)[31:24] = diff4[8:1];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UMAAL

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	0	1	0	0	RdHi				RdLo				Rm				1	0	0	1	Rn			
cond				opc								S																			

Encoding

UMAAL{<c>}{<q>}<RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	1	0	Rn				RdLo				RdHi				0	1	1	0	Rm				
op1																								op2							

Encoding

UMAAL{<c>}{<q>}<RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = UInt(R(n)) * UInt(R(m)) + UInt(R(dHi)) + UInt(R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLAL, UMLALS

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value. In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	0	1	S	RdHi				RdLo				Rm				1	0	0	1	Rn			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

UMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the Not flag setting variant

Applies when (S == 0)

UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	1	1	0	Rn				RdLo				RdHi				0	0	0	0	Rm			
op1																												op2			

Encoding

UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = UInt(R(n)) * UInt(R(m)) + UInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULL, UMULLS

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result. In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	1	0	0	S	RdHi				RdLo				Rm				1	0	0	1	Rn			
cond								opc																							

Encoding for the Flag setting variant

Applies when (S == 1)

UMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the Not flag setting variant

Applies when (S == 0)

UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	0	1	0	Rn				RdLo				RdHi				0	0	0	0	Rm			
op1																												op2			

Encoding

UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q>	See <i>Standard assembler syntax fields</i> .
<RdLo>	Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = UInt(R(n)) * UInt(R(m));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	1	0	Rn				Rd				(1)	(1)	(1)	(1)	0	0	0	1	Rm			
cond				op1																B				op2							

Encoding

UQADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	0	1	0	0	1	Rn				1	1	1	1	Rd				0	1	0	1	Rm							
op1												U																H				S			

Encoding

UQADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[15:0]) + UInt(R(m)[15:0]);
    let sum2 : integer = UInt(R(n)[31:16]) + UInt(R(m)[31:16]);
    R(d)[15:0] = UnsignedSat{16}(sum1);
    R(d)[31:16] = UnsignedSat{16}(sum2);
end;
```

UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	1	0	Rn				Rd				(1)	(1)	(1)	(1)	1	0	0	1	Rm			
cond				op1																B				op2							

Encoding

UQADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	0	1	0	0	0	Rn				1	1	1	1	Rd				0	1	0	1	Rm							
								op1																U				H				S			

Encoding

UQADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[7:0]) + UInt(R(m)[7:0]);
    let sum2 : integer = UInt(R(n)[15:8]) + UInt(R(m)[15:8]);
    let sum3 : integer = UInt(R(n)[23:16]) + UInt(R(m)[23:16]);
    let sum4 : integer = UInt(R(n)[31:24]) + UInt(R(m)[31:24]);
    R(d)[7:0] = UnsignedSat{8}(sum1);
    R(d)[15:8] = UnsignedSat{8}(sum2);
    R(d)[23:16] = UnsignedSat{8}(sum3);
    R(d)[31:24] = UnsignedSat{8}(sum4);
end;
```

UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	1	0	Rn				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rm			
cond				op1																B				op2							

Encoding

UQASX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	0	Rn				1	1	1	1	Rd				0	1	0	1	Rm			
op1												U												H				S			

Encoding

UQASX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = UInt(R(n)[15:0]) - UInt(R(m)[31:16]);
    let sum : integer = UInt(R(n)[31:16]) + UInt(R(m)[15:0]);
    R(d)[15:0] = UnsignedSat{16}(diff);
    R(d)[31:16] = UnsignedSat{16}(sum);
end;
```

UQSAX

Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	1	0	Rn				Rd				(1)	(1)	(1)	(1)	0	1	0	1	Rm			
cond				op1																B				op2							

Encoding

UQSAX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	1	0	1	1	1	0	Rn				1	1	1	1	Rd				0	1	0	1	Rm							
								op1																U				H				S			

Encoding

UQSAX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = UInt(R(n)[15:0]) + UInt(R(m)[31:16]);
    let diff : integer = UInt(R(n)[31:16]) - UInt(R(m)[15:0]);
    R(d)[15:0] = UnsignedSat{16}(sum);
    R(d)[31:16] = UnsignedSat{16}(diff);
end;
```

UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	1	0	Rn				Rd				(1)	(1)	(1)	(1)	0	1	1	1	Rm			
cond				op1																B				op2							

Encoding

UQSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	1	Rn				1	1	1	1	Rd				0	1	0	1	Rm			
op1												U H S																			

Encoding

UQSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

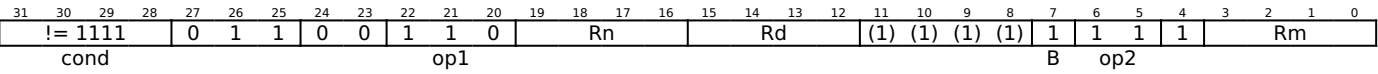
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[15:0]) - UInt(R(m)[15:0]);
    let diff2 : integer = UInt(R(n)[31:16]) - UInt(R(m)[31:16]);
    R(d)[15:0] = UnsignedSat{16}(diff1);
    R(d)[31:16] = UnsignedSat{16}(diff2);
end;
```

UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$, and writes the results to the destination register. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



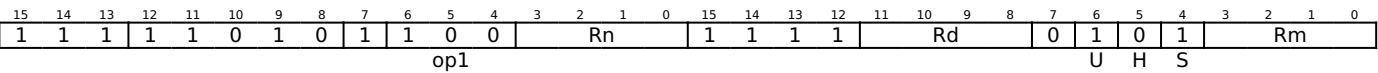
Encoding

UQSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UQSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[7:0]) - UInt(R(m)[7:0]);
    let diff2 : integer = UInt(R(n)[15:8]) - UInt(R(m)[15:8]);
    let diff3 : integer = UInt(R(n)[23:16]) - UInt(R(m)[23:16]);
    let diff4 : integer = UInt(R(n)[31:24]) - UInt(R(m)[31:24]);
    R(d)[7:0] = UnsignedSat{8}(diff1);
    R(d)[15:8] = UnsignedSat{8}(diff2);
    R(d)[23:16] = UnsignedSat{8}(diff3);
    R(d)[31:24] = UnsignedSat{8}(diff4);
end;
```


USAD8

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111								0 1 1				1 1 0 0 0				Rd				1 1 1 1				Rm				0 0 0			1	Rn			
cond																Ra																			

Encoding

USAD8 { <c> } { <q> } { <Rd>, } <Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	1	1	1	Rn				1	1	1	1	Rd				0	0	0	0	Rm			
op1												Ra				op2															

Encoding

$$\text{USAD8} \{ \langle c \rangle \} \{ \langle q \rangle \} \{ \langle Rd \rangle, \} \langle Rn \rangle, \langle Rm \rangle$$

Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```

if ConditionPassed() then
  EncodingSpecificOperations();
  let absdiff1 : integer = Abs(UInt(R(n)[7:0]) - UInt(R(m)[7:0]));
  let absdiff2 : integer = Abs(UInt(R(n)[15:8]) - UInt(R(m)[15:8]));
  let absdiff3 : integer = Abs(UInt(R(n)[23:16]) - UInt(R(m)[23:16]));
  let absdiff4 : integer = Abs(UInt(R(n)[31:24]) - UInt(R(m)[31:24]));
  let result : integer = absdiff1 + absdiff2 + absdiff3 + absdiff4;
  R(d) = result[31:0];
end;

```

2026-03 rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USADA8

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	0	0	0	Rd				!= 1111				Rm				0	0	0	1	Rn			
cond												Ra																			

Encoding

USADA8{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```
if Ra == '1111' then See("USAD8"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	0	1	1	0	1	1	1	Rn				!= 1111				Rd				0		0	0		0	Rm			
op1												Ra																op2					

Encoding

USADA8{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```
if Ra == '1111' then See("USAD8"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let absdiff1 : integer = Abs(UInt(R(n)[7:0]) - UInt(R(m)[7:0]));
    let absdiff2 : integer = Abs(UInt(R(n)[15:8]) - UInt(R(m)[15:8]));
    let absdiff3 : integer = Abs(UInt(R(n)[23:16]) - UInt(R(m)[23:16]));
    let absdiff4 : integer = Abs(UInt(R(n)[31:24]) - UInt(R(m)[31:24]));
    let result : integer = UInt(R(a)) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R(d) = result[31:0];
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

This instruction sets *PSTATE.Q* to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	1	sat_imm					Rd					imm5					sh	0	1	Rn		
cond				U																											

Encoding for the Arithmetic shift right variant

Applies when (sh == 1)

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Encoding for the Logical shift left variant

Applies when (sh == 0)

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(sh::'0', imm5);
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	1	0	sh	0	Rn				0	imm3				Rd				imm2		(0)	sat imm			

Encoding for the Arithmetic shift right variant

Applies when (sh == 1 && !(imm3 == 000 && imm2 == 00))

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Encoding for the Logical shift left variant

Applies when (sh == 0)

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
if sh == '1' && (imm3::imm2) == '0000' then See("USAT16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(sh::'0', imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the bit position for saturation, in the range 0 to 31, encoded in the "sat_imm" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<amount>	For the "A1 Arithmetic shift right" variant: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32. For the "A1 Logical shift left" variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field. For the "T1 Arithmetic shift right" variant: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

For the "T1 Logical shift left" variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  let operand : bits(32) = Shift{}(R(n), shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
  let (result, sat) : (bits(saturate_to), boolean) = UnsignedSatQ{saturate_to}(SInt(operand));
  R(d) = ZeroExtend{32}(result);
  if sat then
    PSTATE.Q = '1';
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range. This instruction sets *PSTATE.Q* to 1 if the operation saturates. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	1	0	sat imm			Rd			(1)	(1)	(1)	(1)	0	0	1	1	Rn					
cond									U																						

Encoding

USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	1	0	1	0	Rn				0	0	0	0	Rd				0	0	(0)	(0)	sat imm			
op1												imm3				imm2															

Encoding

USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <imm> Is the bit position for saturation, in the range 0 to 15, encoded in the "sat_imm" field.
- <Rn> Is the general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 = SInt(R(n)[15:0]);
    let operand2 = SInt(R(n)[31:16]);
    let (result1, sat1) : (bits(saturate_to), boolean) = UnsignedSatQ{saturate_to}(operand1);
    let (result2, sat2) : (bits(saturate_to), boolean) = UnsignedSatQ{saturate_to}(operand2);
    R(d)[15:0] = ZeroExtend{16}(result1);
    R(d)[31:16] = ZeroExtend{16}(result2);
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```

USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets *PSTATE*.GE according to the results. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	0	1	Rn				Rd				(1)	(1)	(1)	(1)	0	1	0	1	Rm			
cond				op1																B				op2							

Encoding

USAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	1	0	Rn				1	1	1	1	Rd				0	1	0	0	Rm			
op1												U H S																			

Encoding

USAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = UInt(R(n)[15:0]) + UInt(R(m)[31:16]);
    let diff : integer = UInt(R(n)[31:16]) - UInt(R(m)[15:0]);
    R(d)[15:0] = sum[15:0];
    R(d)[31:16] = diff[15:0];
    PSTATE.GE[1:0] = if sum >= 0x10000 then '11' else '00';
    PSTATE.GE[3:2] = if diff >= 0 then '11' else '00';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	0	1	Rn				Rd				(1)	(1)	(1)	(1)	0	1	1	1	Rm			
cond				op1												B				op2											

Encoding

USUB16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	1	Rn				1	1	1	1	Rd				0	1	0	0	Rm			
op1												U												H				S			

Encoding

USUB16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the *CONSTRAINED UNPREDICTABLE* behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[15:0]) - UInt(R(m)[15:0]);
    let diff2 : integer = UInt(R(n)[31:16]) - UInt(R(m)[31:16]);
    R(d)[15:0] = diff1[15:0];
    R(d)[31:16] = diff2[15:0];
    PSTATE.GE[1:0] = if diff1 >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if diff2 >= 0 then '11' else '00';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	1	0	1	Rn				Rd				(1)	(1)	(1)	(1)	1	1	1	1	Rm			
cond				op1																B				op2							

Encoding

USUB8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	0	0	Rn				1	1	1	1	Rd				0	1	0	0	Rm			
op1												U H S																			

Encoding

USUB8{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[7:0]) - UInt(R(m)[7:0]);
    let diff2 : integer = UInt(R(n)[15:8]) - UInt(R(m)[15:8]);
    let diff3 : integer = UInt(R(n)[23:16]) - UInt(R(m)[23:16]);
    let diff4 : integer = UInt(R(n)[31:24]) - UInt(R(m)[31:24]);
    R(d)[7:0] = diff1[7:0];
    R(d)[15:8] = diff2[7:0];
    R(d)[23:16] = diff3[7:0];
    R(d)[31:24] = diff4[7:0];
    PSTATE.GE[0] = if diff1 >= 0 then '1' else '0';
    PSTATE.GE[1] = if diff2 >= 0 then '1' else '0';
    PSTATE.GE[2] = if diff3 >= 0 then '1' else '0';
    PSTATE.GE[3] = if diff4 >= 0 then '1' else '0';
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

UXTAB

Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	0	!= 1111				Rd				rotate	(0)	(0)	0	1	1	1	Rm					
cond				U				op				Rn																			

Encoding

UXTAB{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	1	!= 1111				1	1	1	1	Rd				1	(0)	rotate	Rm				
op1												U				Rn															

Encoding

UXTAB{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.
<amount>	Is the rotate amount, encoded in "rotate":

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + ZeroExtend{32}(rotated[7:0]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UXTAB16

Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	0	0	!= 1111				Rd				rotate		(0)	(0)	0	1	1	1	Rm			
cond				U				op				Rn																			

Encoding

UXTAB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	1	!= 1111				1	1	1	1	Rd				1	(0)	rotate	Rm				
op1												U				Rn															

Encoding

UXTAB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<amount>	Is the rotate amount, encoded in “rotate”:										
<table><tr><th>rotate</th><th><amount></th></tr><tr><td>00</td><td>[absent]</td></tr><tr><td>01</td><td>8</td></tr><tr><td>10</td><td>16</td></tr><tr><td>11</td><td>24</td></tr></table>		rotate	<amount>	00	[absent]	01	8	10	16	11	24
rotate	<amount>										
00	[absent]										
01	8										
10	16										
11	24										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0] = R(n)[15:0] + ZeroExtend{16}(rotated[7:0]);
    R(d)[31:16] = R(n)[31:16] + ZeroExtend{16}(rotated[23:16]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTAH

Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	1	1	!= 1111				Rd				rotate		(0)	(0)	0	1	1	1	Rm			
cond				U				op				Rn																			

Encoding

UXTAH{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	1	!= 1111				1	1	1	1	Rd				1	(0)	rotate	Rm				
op1												U				Rn															

Encoding

UXTAH{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.										
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.										
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.										
<amount>	Is the rotate amount, encoded in “rotate”:										
<table><tr><th>rotate</th><th><amount></th></tr><tr><td>00</td><td>[absent]</td></tr><tr><td>01</td><td>8</td></tr><tr><td>10</td><td>16</td></tr><tr><td>11</td><td>24</td></tr></table>		rotate	<amount>	00	[absent]	01	8	10	16	11	24
rotate	<amount>										
00	[absent]										
01	8										
10	16										
11	24										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + ZeroExtend{32}(rotated[15:0]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm				
cond				U				op				Rn																			

Encoding

UXTB{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	1	0	0	1	0	1	1	Rm			Rd			
								U	B							

Encoding

UXTB{<c>}{<q>} {<Rd>, }<Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate	Rm				
op1																U				Rn											

Encoding

UXTB{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

UXTB{<c>}.W {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = ZeroExtend{32}(rotated[7:0]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTB16

Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	0	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			
cond				U				op				Rn																			

Encoding

UXTB16{<c>}{<q>}{<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
1	1	1	1	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate	Rm																				
op1																U																Rn															

Encoding

UXTB16{<c>}{<q>}{<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> For the "A1" variant: is the general-purpose source register, encoded in the "Rm" field.
For the "T1" variant: is the second general-purpose source register, encoded in the "Rm" field.
- <amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0] = ZeroExtend{16}(rotated[7:0]);
    R(d)[31:16] = ZeroExtend{16}(rotated[23:16]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	1	1	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			
cond				U				op				Rn																			

Encoding

UXTH{<c>}{<q>}{<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	0	Rm			Rd		
								U	B						

Encoding

UXTH{<c>}{<q>}{<Rd>, }<Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate	Rm				
op1																U				Rn											

Encoding

UXTH{<c>}{<q>}{<Rd>, }<Rm> {, ROR #<amount>}

UXTH{<c>}.W {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = ZeroExtend{32}(rotated[15:0]);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event and Send Event*.

As described in *Wait For Event and Send Event*, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see HCR.TWE, SCR.TWE, and SCTLR.nTWE.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	0
cond				R								imm4				imm12															

Encoding

WFE{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0
hint															

Encoding

WFE{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	0
hint																option															

Encoding

WFE{<c>}.W

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if IsEventRegisterSet() then
        ClearEventRegister();
    else
        if HaveEL(EL3) && EL3SDDUndefPriority() then
            // Check for traps described by the Secure Monitor.
            // If the trap is enabled, the instruction will be UNDEFINED because EDSCR.SDD is 1.
            AArch32\_CheckForWfxTrap(EL3, WfxType\_WFE);
        end;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32\_CheckForWfxTrap(EL1, WfxType\_WFE);
        end;
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32\_CheckForWfxTrap(EL2, WfxType\_WFE);
        end;
        if HaveEL(EL3) && PSTATE.M != M32\_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32\_CheckForWfxTrap(EL3, WfxType\_WFE);
        end;
        WaitForEvent();
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see *Wait For Interrupt*.
As described in *Wait For Interrupt*, the execution of a WFI instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see *HCR.TWI*, *SCR.TWI*, and *SCTLR.nTWI*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	1
cond				R				imm4				imm12																			

Encoding

WFI{<c>} {<q>}

Decode for this encoding

// No additional decoding required

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0
hint															

Encoding

WFI{<c>} {<q>}

Decode for this encoding

// No additional decoding required

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	1
hint																option															

Encoding

WFI{<c>}.W

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !AArch32 InterruptPending() then
        if HaveEL(EL3) && EL3SDDUndefPriority() then
            // Check for traps described by the Secure Monitor.
            // If the trap is enabled, the instruction will be UNDEFINED because EDSCR.SDD is 1.
            AArch32 CheckForWfxTrap(EL3, WfxType WFI);
        end;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32 CheckForWfxTrap(EL1, WfxType WFI);
        end;
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32 CheckForWfxTrap(EL2, WfxType WFI);
        end;
        if HaveEL(EL3) && PSTATE.M != M32 Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32 CheckForWfxTrap(EL3, WfxType WFI);
        end;
        WaitForInterrupt();
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction see *The Yield instruction*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	1
cond				R								imm4				imm12															

Encoding

YIELD{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0
hint															

Encoding

YIELD{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	1
hint																option															

Encoding

YIELD{<c>}.W

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c>

See *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
end;
```


AArch32 -- SIMD&FP Instructions (alphabetic order)

[AESD](#): AES single round decryption.

[AESE](#): AES single round encryption.

[AESIMC](#): AES inverse mix columns.

[AESMC](#): AES mix columns.

[FLDM*X \(FLDMDBX, FLDMIAX\)](#): FLDM*X.

[FSTMDBX, FSTMIAX](#): FSTMX.

[SHA1C](#): SHA1 hash update (choose).

[SHA1H](#): SHA1 fixed rotate.

[SHA1M](#): SHA1 hash update (majority).

[SHA1P](#): SHA1 hash update (parity).

[SHA1SU0](#): SHA1 schedule update 0.

[SHA1SU1](#): SHA1 schedule update 1.

[SHA256H](#): SHA256 hash update part 1.

[SHA256H2](#): SHA256 hash update part 2.

[SHA256SU0](#): SHA256 schedule update 0.

[SHA256SU1](#): SHA256 schedule update 1.

[VABA](#): Vector Absolute Difference and Accumulate.

[VABAL](#): Vector Absolute Difference and Accumulate Long.

[VABD \(floating-point\)](#): Vector Absolute Difference (floating-point).

[VABD \(integer\)](#): Vector Absolute Difference (integer).

[VABDL \(integer\)](#): Vector Absolute Difference Long (integer).

[VABS](#): Vector Absolute.

[VACGE](#): Vector Absolute Compare Greater Than or Equal.

[VACGT](#): Vector Absolute Compare Greater Than.

[VACLE](#): Vector Absolute Compare Less Than or Equal: an alias of VACGE.

[VACLT](#): Vector Absolute Compare Less Than: an alias of VACGT.

[VADD \(floating-point\)](#): Vector Add (floating-point).

[VADD \(integer\)](#): Vector Add (integer).

[VADDHN](#): Vector Add and Narrow, returning High Half.

[VADDL](#): Vector Add Long.

[VADDW](#): Vector Add Wide.

[VAND \(immediate\)](#): Vector Bitwise AND (immediate): an alias of VBIC (immediate).

[VAND \(register\)](#): Vector Bitwise AND (register).

[VBIC \(immediate\)](#): Vector Bitwise Bit Clear (immediate).

[VBIC \(register\)](#): Vector Bitwise Bit Clear (register).

[VBIF](#): Vector Bitwise Insert if False.

[VBIT](#): Vector Bitwise Insert if True.

[VBSL](#): Vector Bitwise Select.

[VCADD](#): Vector Complex Add.

[VCEQ \(immediate #0\)](#): Vector Compare Equal to Zero.

[VCEQ \(register\)](#): Vector Compare Equal.

[VCGE \(immediate #0\)](#): Vector Compare Greater Than or Equal to Zero.

[VCGE \(register\)](#): Vector Compare Greater Than or Equal.

[VCGT \(immediate #0\)](#): Vector Compare Greater Than Zero.

[VCGT \(register\)](#): Vector Compare Greater Than.

[VCLE \(immediate #0\)](#): Vector Compare Less Than or Equal to Zero.

[VCLE \(register\)](#): Vector Compare Less Than or Equal: an alias of VCGE (register).

[VCLS](#): Vector Count Leading Sign Bits.

[VCLT \(immediate #0\)](#): Vector Compare Less Than Zero.

[VCLT \(register\)](#): Vector Compare Less Than: an alias of VCGT (register).

[VCLZ](#): Vector Count Leading Zeros.

[VCMLA](#): Vector Complex Multiply Accumulate.

[VCMLA \(by element\)](#): Vector Complex Multiply Accumulate (by element).

[VCMP](#): Vector Compare.

[VCMPPE](#): Vector Compare, raising Invalid Operation on NaN.

[VCNT](#): Vector Count Set Bits.

[VCVT \(between double-precision and single-precision\)](#): Convert between double-precision and single-precision.

[VCVT \(between floating-point and fixed-point, Advanced SIMD\)](#): Vector Convert between floating-point and fixed-point.

[VCVT \(between floating-point and fixed-point, floating-point\)](#): Convert between floating-point and fixed-point.

[VCVT \(between floating-point and integer, Advanced SIMD\)](#): Vector Convert between floating-point and integer.

[VCVT \(between half-precision and single-precision, Advanced SIMD\)](#): Vector Convert between half-precision and single-precision.

[VCVT \(floating-point to integer, floating-point\)](#): Convert floating-point to integer with Round towards Zero.

[VCVT \(from single-precision to BFloat16, Advanced SIMD\)](#): Vector Convert from single-precision to BFloat16.

[VCVT \(integer to floating-point, floating-point\)](#): Convert integer to floating-point.

[VCVTA \(Advanced SIMD\)](#): Vector Convert floating-point to integer with Round to Nearest with Ties to Away.

[VCVTA \(floating-point\)](#): Convert floating-point to integer with Round to Nearest with Ties to Away.

[VCVTB](#): Convert to or from a half-precision value in the bottom half of a single-precision register.

[VCVTB \(BFloat16\)](#): Converts from a single-precision value to a BFloat16 value in the bottom half of a single-precision register.

[VCVTM \(Advanced SIMD\)](#): Vector Convert floating-point to integer with Round towards -Infinity.

[VCVTM \(floating-point\)](#): Convert floating-point to integer with Round towards -Infinity.

[VCVTN \(Advanced SIMD\)](#): Vector Convert floating-point to integer with Round to Nearest.

[VCVTN \(floating-point\)](#): Convert floating-point to integer with Round to Nearest.

[VCVTP \(Advanced SIMD\)](#): Vector Convert floating-point to integer with Round towards +Infinity.

[VCVTP \(floating-point\)](#): Convert floating-point to integer with Round towards +Infinity.

[VCVTR](#): Convert floating-point to integer.

[VCVTT](#): Convert to or from a half-precision value in the top half of a single-precision register.

[VCVTT \(BFloat16\)](#): Converts from a single-precision value to a BFloat16 value in the top half of a single-precision register.

[VDIV](#): Divide.

[VDOT \(by element\)](#): BFloat16 floating-point indexed dot product (vector, by element).

[VDOT \(vector\)](#): BFloat16 floating-point (BF16) dot product (vector).

[VDUP \(general-purpose register\)](#): Duplicate general-purpose register to vector.

[VDUP \(scalar\)](#): Duplicate vector element to vector.

[VEOR](#): Vector Bitwise Exclusive-OR.

[VEXT \(byte elements\)](#): Vector Extract.

[VEXT \(multibyte elements\)](#): Vector Extract: an alias of VEXT (byte elements).

[VFMA](#): Vector Fused Multiply Accumulate.

[VFMA, VFMA, VFMA \(BFloat16, by scalar\)](#): BFloat16 floating-point widening multiply-add long (by scalar).

[VFMA, VFMA, VFMA \(BFloat16, vector\)](#): BFloat16 floating-point widening multiply-add long (vector).

[VFMA \(by scalar\)](#): Vector Floating-point Multiply-Add Long to accumulator (by scalar).

[VFMA \(vector\)](#): Vector Floating-point Multiply-Add Long to accumulator (vector).

[VFMS](#): Vector Fused Multiply Subtract.

[VFMSL \(by scalar\)](#): Vector Floating-point Multiply-Subtract Long from accumulator (by scalar).

[VFMSL \(vector\)](#): Vector Floating-point Multiply-Subtract Long from accumulator (vector).

[VFNMA](#): Vector Fused Negate Multiply Accumulate.

[VFNMS](#): Vector Fused Negate Multiply Subtract.

[VHADD](#): Vector Halving Add.

[VHSUB](#): Vector Halving Subtract.

[VINS](#): Vector move Insertion.

[VJCVT](#): JavaScript Convert to signed fixed-point, rounding toward Zero.

[VLD1 \(multiple single elements\)](#): Load multiple single 1-element structures to one, two, three, or four registers.

[VLD1 \(single element to all lanes\)](#): Load single 1-element structure and replicate to all lanes of one register.

[VLD1 \(single element to one lane\)](#): Load single 1-element structure to one lane of one register.

[VLD2 \(multiple 2-element structures\)](#): Load multiple 2-element structures to two or four registers.

[VLD2 \(single 2-element structure to all lanes\)](#): Load single 2-element structure and replicate to all lanes of two registers.

[VLD2 \(single 2-element structure to one lane\)](#): Load single 2-element structure to one lane of two registers.

[VLD3 \(multiple 3-element structures\)](#): Load multiple 3-element structures to three registers.

[VLD3 \(single 3-element structure to all lanes\)](#): Load single 3-element structure and replicate to all lanes of three registers.

[VLD3 \(single 3-element structure to one lane\)](#): Load single 3-element structure to one lane of three registers.

[VLD4 \(multiple 4-element structures\)](#): Load multiple 4-element structures to four registers.

[VLD4 \(single 4-element structure to all lanes\)](#): Load single 4-element structure and replicate to all lanes of four registers.

[VLD4 \(single 4-element structure to one lane\)](#): Load single 4-element structure to one lane of four registers.

[VLDM, VLDMDB, VLDMIA](#): Load Multiple SIMD&FP registers.

[VLDR \(immediate\)](#): Load SIMD&FP register (immediate).

[VLDR \(literal\)](#): Load SIMD&FP register (literal).

[VMAX \(floating-point\)](#): Vector Maximum (floating-point).

[VMAX \(integer\)](#): Vector Maximum (integer).

[VMAXNM](#): Floating-point Maximum Number.

[VMIN \(floating-point\)](#): Vector Minimum (floating-point).

[VMIN \(integer\)](#): Vector Minimum (integer).

[VMINNM](#): Floating-point Minimum Number.

[VMLA \(by scalar\)](#): Vector Multiply Accumulate (by scalar).

[VMLA \(floating-point\)](#): Vector Multiply Accumulate (floating-point).

[VMLA \(integer\)](#): Vector Multiply Accumulate (integer).

[VMLAL \(by scalar\)](#): Vector Multiply Accumulate Long (by scalar).

[VMLAL \(integer\)](#): Vector Multiply Accumulate Long (integer).

[VMLS \(by scalar\)](#): Vector Multiply Subtract (by scalar).

[VMLS \(floating-point\)](#): Vector Multiply Subtract (floating-point).

[VMLS \(integer\)](#): Vector Multiply Subtract (integer).

[VMLSL \(by scalar\)](#): Vector Multiply Subtract Long (by scalar).

[VMLSL \(integer\)](#): Vector Multiply Subtract Long (integer).

[VMMLA](#): BFloat16 floating-point matrix multiply-accumulate.

[VMOV \(between general-purpose register and half-precision\)](#): Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register.

[VMOV \(between general-purpose register and single-precision\)](#): Copy a general-purpose register to or from a 32-bit SIMD&FP register.

[VMOV \(between two general-purpose registers and a doubleword floating-point register\)](#): Copy two general-purpose registers to or from a SIMD&FP register.

[VMOV \(between two general-purpose registers and two single-precision registers\)](#): Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers.

[VMOV \(general-purpose register to scalar\)](#): Copy a general-purpose register to a vector element.

[VMOV \(immediate\)](#): Copy immediate value to a SIMD&FP register.

[VMOV \(register\)](#): Copy between FP registers.

[VMOV \(register, SIMD\)](#): Copy between SIMD registers: an alias of VORR (register).

[VMOV \(scalar to general-purpose register\)](#): Copy a vector element to a general-purpose register with sign or zero extension.

[VMOVL](#): Vector Move Long.

[VMOVN](#): Vector Move and Narrow.

[VMOVX](#): Vector Move extraction.

[VMRS](#): Move SIMD&FP Special register to general-purpose register.

[VMSR](#): Move general-purpose register to SIMD&FP Special register.

[VMUL \(by scalar\)](#): Vector Multiply (by scalar).

[VMUL \(floating-point\)](#): Vector Multiply (floating-point).

[VMUL \(integer and polynomial\)](#): Vector Multiply (integer and polynomial).

[VMULL \(by scalar\)](#): Vector Multiply Long (by scalar).

[VMULL \(integer and polynomial\)](#): Vector Multiply Long (integer and polynomial).

[VMVN \(immediate\)](#): Vector Bitwise NOT (immediate).

[VMVN \(register\)](#): Vector Bitwise NOT (register).

[VNEG](#): Vector Negate.

[VNMLA](#): Vector Negate Multiply Accumulate.

[VNMLS](#): Vector Negate Multiply Subtract.

[VNMUL](#): Vector Negate Multiply.

[VORN \(immediate\)](#): Vector Bitwise OR NOT (immediate): an alias of VORR (immediate).

[VORN \(register\)](#): Vector bitwise OR NOT (register).

[VORR \(immediate\)](#): Vector Bitwise OR (immediate).

[VORR \(register\)](#): Vector bitwise OR (register).

[VPADAL](#): Vector Pairwise Add and Accumulate Long.

[VPADD \(floating-point\)](#): Vector Pairwise Add (floating-point).

[VPADD \(integer\)](#): Vector Pairwise Add (integer).

[VPADDL](#): Vector Pairwise Add Long.

[VPMAX \(floating-point\)](#): Vector Pairwise Maximum (floating-point).

[VPMAX \(integer\)](#): Vector Pairwise Maximum (integer).

[VPMIN \(floating-point\)](#): Vector Pairwise Minimum (floating-point).

[VPMIN \(integer\)](#): Vector Pairwise Minimum (integer).

[VPOP](#): Pop SIMD&FP registers from stack: an alias of VLDM, VLDMDB, VLDMIA.

[VPUSH](#): Push SIMD&FP registers to stack: an alias of VSTM, VSTMDB, VSTMIA.

[VQABS](#): Vector Saturating Absolute.

[VQADD](#): Vector Saturating Add.

[VQDMLAL](#): Vector Saturating Doubling Multiply Accumulate Long.

[VQDMLSL](#): Vector Saturating Doubling Multiply Subtract Long.

[VQDMULH](#): Vector Saturating Doubling Multiply Returning High Half.

[VQDMULL](#): Vector Saturating Doubling Multiply Long.

[VQMOVN, VQMOVUN](#): Vector Saturating Move and Narrow.

[VQNEG](#): Vector Saturating Negate.

[VQRDMLAH](#): Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half.

[VQRDMLSH](#): Vector Saturating Rounding Doubling Multiply Subtract Returning High Half.

[VQRDMULH](#): Vector Saturating Rounding Doubling Multiply Returning High Half.

[VQRSHL](#): Vector Saturating Rounding Shift Left.

[VQRSHRN \(zero\)](#): Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

[VQRSHRN, VQRSHRUN](#): Vector Saturating Rounding Shift Right, Narrow.

[VQRSHRUN \(zero\)](#): Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

[VQSHL \(register\)](#): Vector Saturating Shift Left (register).

[VQSHL, VQSHLU \(immediate\)](#): Vector Saturating Shift Left (immediate).

[VQSHRN \(zero\)](#): Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

[VQSHRN, VQSHRUN](#): Vector Saturating Shift Right, Narrow.

[VQSHRUN \(zero\)](#): Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

[VQSUB](#): Vector Saturating Subtract.

[VRADDHN](#): Vector Rounding Add and Narrow, returning High Half.

[VRECPE](#): Vector Reciprocal Estimate.

[VRECPS](#): Vector Reciprocal Step.

[VREV16](#): Vector Reverse in halfwords.

[VREV32](#): Vector Reverse in words.

[VREV64](#): Vector Reverse in doublewords.

[VRHADD](#): Vector Rounding Halving Add.

[VRINTA \(Advanced SIMD\)](#): Vector Round floating-point to integer towards Nearest with Ties to Away.

[VRINTA \(floating-point\)](#): Round floating-point to integer to Nearest with Ties to Away.

[VRINTM \(Advanced SIMD\)](#): Vector Round floating-point to integer towards -Infinity.

[VRINTM \(floating-point\)](#): Round floating-point to integer towards -Infinity.

[VRINTN \(Advanced SIMD\)](#): Vector Round floating-point to integer to Nearest.

[VRINTN \(floating-point\)](#): Round floating-point to integer to Nearest.

[VRINTP \(Advanced SIMD\)](#): Vector Round floating-point to integer towards +Infinity.

[VRINTP \(floating-point\)](#): Round floating-point to integer towards +Infinity.

[VRINTR](#): Round floating-point to integer.

[VRINTX \(Advanced SIMD\)](#): Vector round floating-point to integer inexact.

[VRINTX \(floating-point\)](#): Round floating-point to integer inexact.

[VRINTZ \(Advanced SIMD\)](#): Vector round floating-point to integer towards Zero.

[VRINTZ \(floating-point\)](#): Round floating-point to integer towards Zero.

[VRSHL](#): Vector Rounding Shift Left.

[VRSHR](#): Vector Rounding Shift Right.

[VRSHR \(zero\)](#): Vector Rounding Shift Right: an alias of VORR (register).

[VRSHRN](#): Vector Rounding Shift Right and Narrow.

[VRSHRN \(zero\)](#): Vector Rounding Shift Right and Narrow: an alias of VMOVN.

[VSQRTE](#): Vector Reciprocal Square Root Estimate.

[VSQRTS](#): Vector Reciprocal Square Root Step.

[VSRRA](#): Vector Rounding Shift Right and Accumulate.

[VRSUBHN](#): Vector Rounding Subtract and Narrow, returning High Half.

[VSDOT \(by element\)](#): Dot Product index form with signed integers.

[VSDOT \(vector\)](#): Dot Product vector form with signed integers.

[VSELEQ, VSELGE, VSELGT, VSELVS](#): Floating-point conditional select.

[VSHL \(immediate\)](#): Vector Shift Left (immediate).

[VSHL \(register\)](#): Vector Shift Left (register).

[VSHLL](#): Vector Shift Left Long.

[VSHR](#): Vector Shift Right.

[VSHR \(zero\)](#): Vector Shift Right: an alias of VORR (register).

[VSHRN](#): Vector Shift Right Narrow.

[VSHRN \(zero\)](#): Vector Shift Right Narrow: an alias of VMOVN.

[VSLI](#): Vector Shift Left and Insert.

[VSMMLA](#): Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix.

[VSQRT](#): Square Root.

[VSRA](#): Vector Shift Right and Accumulate.

[VSRI](#): Vector Shift Right and Insert.

[VST1 \(multiple single elements\)](#): Store multiple single elements from one, two, three, or four registers.

[VST1 \(single element from one lane\)](#): Store single element from one lane of one register.

[VST2 \(multiple 2-element structures\)](#): Store multiple 2-element structures from two or four registers.

[VST2 \(single 2-element structure from one lane\)](#): Store single 2-element structure from one lane of two registers.

[VST3 \(multiple 3-element structures\)](#): Store multiple 3-element structures from three registers.

[VST3 \(single 3-element structure from one lane\)](#): Store single 3-element structure from one lane of three registers.

[VST4 \(multiple 4-element structures\)](#): Store multiple 4-element structures from four registers.

[VST4 \(single 4-element structure from one lane\)](#): Store single 4-element structure from one lane of four registers.

[VSTM, VSTMDB, VSTMIA](#): Store multiple SIMD&FP registers.

[VSTR](#): Store SIMD&FP register.

[VSUB \(floating-point\)](#): Vector Subtract (floating-point).

[VSUB \(integer\)](#): Vector Subtract (integer).

[VSUBHN](#): Vector Subtract and Narrow, returning High Half.

[VSUBL](#): Vector Subtract Long.

[VSUBW](#): Vector Subtract Wide.

[VSUDOT \(by element\)](#): Dot Product index form with signed and unsigned integers (by element).

[VSWP](#): Vector Swap.

[VTBL, VTBX](#): Vector Table Lookup and Extension.

[VTRN](#): Vector Transpose.

[VTST](#): Vector Test Bits.

[VUDOT \(by element\)](#): Dot Product index form with unsigned integers.

[VUDOT \(vector\)](#): Dot Product vector form with unsigned integers.

[VUMMLA](#): Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix.

[VUSDOT \(by element\)](#): Dot Product index form with unsigned and signed integers (by element).

[VUSDOT \(vector\)](#): Dot Product vector form with mixed-sign integers.

[VUSMMLA](#): Widening 8-bit mixed integer matrix multiply-accumulate into 2x2 matrix.

[VUZP](#): Vector Unzip.

[VUZP \(alias\)](#): Vector Unzip: an alias of VTRN.

[VZIP](#): Vector Zip.

[VZIP \(alias\)](#): Vector Zip: an alias of VTRN.

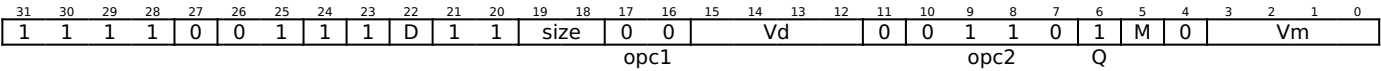
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESD

AES single round decryption.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_AES)



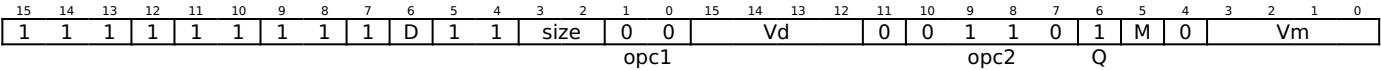
Encoding

AESD.<dt> <Qd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_AES)



Encoding

AESD.<dt> <Qd>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckCryptoEnabled32();
  let operand1 : bits(128) = Q(d>>1);
  let operand2 : bits(128) = Q(m>>1);
  Q(d>>1) = AESInvSubBytes(AESInvShiftRows(operand1 XOR operand2));
end;
```

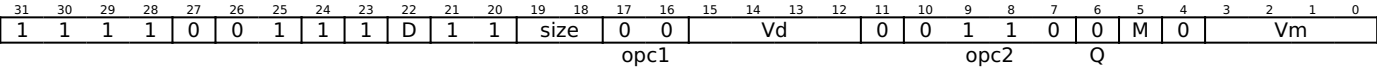
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

AESE

AES single round encryption.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_AES)



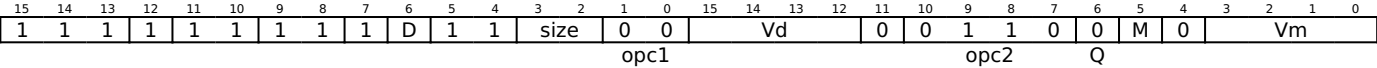
Encoding

AESE.<dt> <Qd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_AES)



Encoding

AESE.<dt> <Qd>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckCryptoEnabled32();
  let operand1 : bits(128) = Q(d>>1);
  let operand2 : bits(128) = Q(m>>1);
  Q(d>>1) = AESSubBytes(AESShiftRows(operand1 XOR operand2));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

AESIMC

AES inverse mix columns.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_AES)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	0	Vd				0	0	opc2				1	M	0	Vm			
opc1																		Q													

Encoding

AESIMC.<dt> <Qd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_AES)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0	Vd				0	0	1	1	1	1	M	0	Vm				
opc1												opc2						Q													

Encoding

AESIMC.<dt> <Qd>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<dt>	Is the data type, encoded in “size”:								
<table><tr><th>size</th><th><dt></th></tr><tr><td>00</td><td>8</td></tr><tr><td>01</td><td>RESERVED</td></tr><tr><td>1x</td><td>RESERVED</td></tr></table>		size	<dt>	00	8	01	RESERVED	1x	RESERVED
size	<dt>								
00	8								
01	RESERVED								
1x	RESERVED								
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.								
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.								

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q(d>>1) = AESInvMixColumns(Q(m>>1));
end;
```

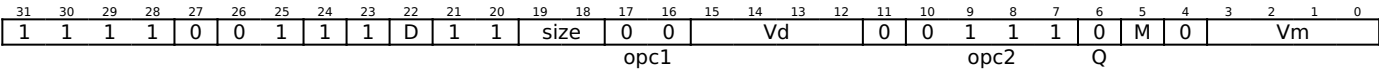
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

AESMC

AES mix columns.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_AES)



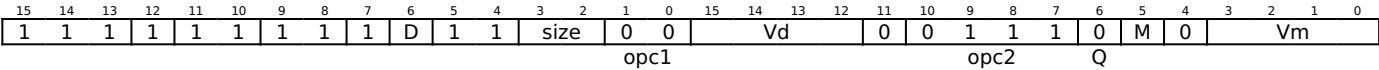
Encoding

AESMC.<dt> <Qd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_AES)



Encoding

AESMC.<dt> <Qd>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<dt>	Is the data type, encoded in “size”:								
<table><tr><th>size</th><th><dt></th></tr><tr><td>00</td><td>8</td></tr><tr><td>01</td><td>RESERVED</td></tr><tr><td>1x</td><td>RESERVED</td></tr></table>	size	<dt>	00	8	01	RESERVED	1x	RESERVED	
size	<dt>								
00	8								
01	RESERVED								
1x	RESERVED								
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.								
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.								

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q(d>>1) = AESMixColumns(Q(m>>1));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

FLDM*X (FLDMDBX, FLDMIAX)

FLDMDBX is the Decrement Before variant of this instruction, and FLDMIAX is the Increment After variant. FLDM*X loads multiple SIMD&FP registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register. Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	1	x	x	x	x	x	x	x	1
cond				L												size				imm8											

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
FLDMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet\(\) != InstrSet\_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	1	x	x	x	x	x	x	x	1
												L								size				imm8							

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
FLDMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

FLDMIAX{<c>}{<q>}<Rn>{!}, <dreglist>

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: See [Advanced SIMD and floating-point 64-bit move](#) for the T32 instruction set, or [Advanced SIMD and floating-point 64-bit move](#) for the A32 instruction set.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.
<dreglist>	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.
!	Specifies base register writeback, encoded in "W":

W	!
0	[absent]
1	[present]

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;

    for r = 0 to regs-1 do
        if single_regs then
            S(d+r) = MemA{32}(address);
            address = address+4;
        else
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            address = address+8;

            // Combine the word-aligned words in the correct order for current endianness.
            D(d+r) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
        end;
    end;

    if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSTMDBX, FSTMIAX

FSTMX stores multiple SIMD&FP registers from the Advanced SIMD and floating-point register file to consecutive locations in using an address from a general-purpose register.

Arm deprecates use of FSTMDBX and FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the *CPACR*, *NSACR*, *HCPtr*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	P	U	D	W	0	Rn				Vd				1	0	1	1	x	x	x	x	x	x	x	1
cond				L												size				imm8											

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
FSTMDBX{<c>}{<q>}<Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
FSTMIAX{<c>}{<q>}<Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet\_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a *VSTM* with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	0	Rn				Vd				1	0	1	1	x	x	x	x	x	x	x	1
												L								size				imm8							

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
FSTMDBX{<c>}{<q>}<Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

FSTMIAX{<c>}{<q>}<Rn>{!}, <dreglist>

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).
Related encodings: See [Advanced SIMD and floating-point 64-bit move](#) for the T32 instruction set, or [Advanced SIMD and floating-point 64-bit move](#) for the A32 instruction set.

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.						
<dreglist>	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.						
!	Specifies base register writeback, encoded in "W": <table><tr><th>W</th><th>!</th></tr><tr><td>0</td><td>[absent]</td></tr><tr><td>1</td><td>[present]</td></tr></table>	W	!	0	[absent]	1	[present]
W	!						
0	[absent]						
1	[present]						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;
    for r = 0 to regs-1 do
        if single_regs then
            MemA{32}(address) = S(d+r);
            address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType_ASIMD) then
                MemA{32}(address) = D(d+r)[63:32];
                MemA{32}(address+4) = D(d+r)[31:0];
            else
                MemA{32}(address) = D(d+r)[31:0];
                MemA{32}(address+4) = D(d+r)[63:32];
            end;
            address = address+8;
        end;
    end;
    if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```


Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

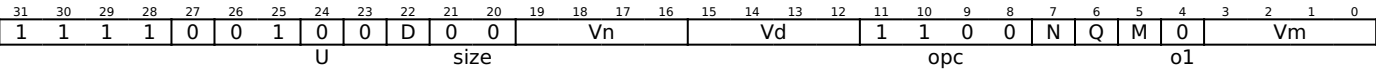
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1C

SHA1 hash update (choose).
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_SHA1)



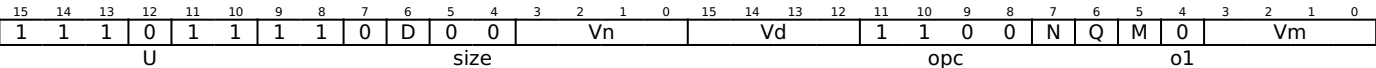
Encoding

SHA1C.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_SHA1)



Encoding

SHA1C.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckCryptoEnabled32();
  var x : bits(128) = Q(d>>1);
  var y : bits(32) = Q(n>>1)[31:0]; // Note: 32 bits wide
  let w : bits(128) = Q(m>>1);
  for e = 0 to 3 do
    let t : bits(32) = SHAchoose(x[63:32], x[95:64], x[127:96]);
    y = y + ROL(x[31:0], 5) + t + w[e*32];
    x[63:32] = ROL(x[63:32], 30);
    let yx : bits(160) = ROL(y::x, 32);
    (y, x) = (yx[128+32], yx[0+128]);
  end;
  Q(d>>1) = x;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHA1H

SHA1 fixed rotate.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1 (FEAT_SHA1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	0	1	0	1	1	M	0	Vm			
opc1												opc2						Q													

Encoding

SHA1H.32 [<Qd>](#), [<Qm>](#)

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1 (FEAT_SHA1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			0	1	Vd				0	0	1	0	1	1	M	0	Vm			
opc1															opc2						Q											

Encoding

SHA1H.32 [<Qd>](#), [<Qm>](#)

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- [<Qd>](#) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as [<Qd>*2](#).
- [<Qm>](#) Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as [<Qm>*2](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    Q(d>>1) = ZeroExtend{128}(ROL(Q(m>>1)[31:0], 30));
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHA1M

SHA1 hash update (majority).
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1 (FEAT_SHA1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	0	1	0	0	D	1	0	Vn				Vd				1				1	0	0	N	Q	M	0	Vm			
U										size				opc								o1												

Encoding

SHA1M.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1 (FEAT_SHA1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	0	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			
U										size				opc								o1									

Encoding

SHA1M.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var x : bits(128) = Q(d>>1);
    var y : bits(32) = Q(n>>1)[31:0]; // Note: 32 bits wide
    let w : bits(128) = Q(m>>1);
    for e = 0 to 3 do
        let t : bits(32) = SHAmajority{(x[63:32], x[95:64], x[127:96]);
        y = y + ROL(x[31:0], 5) + t + w[e*32];
        x[63:32] = ROL(x[63:32], 30);
        let yx : bits(160) = ROL(y::x, 32);
        (y, x) = (yx[128+32], yx[0+128]);
    end;
    Q(d>>1) = x;
end;
```

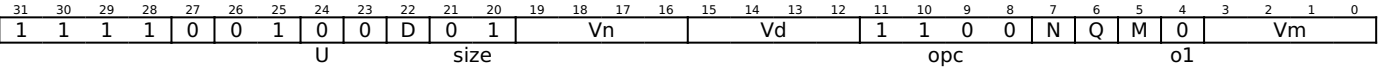
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHA1P

SHA1 hash update (parity).
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1 (FEAT_SHA1)



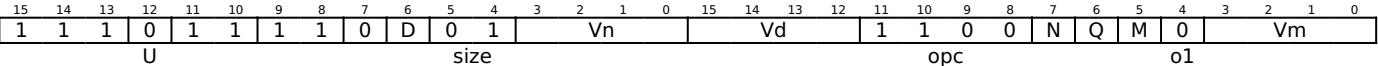
Encoding

SHA1P.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1 (FEAT_SHA1)



Encoding

SHA1P.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- [<Qd>](#) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as [<Qd>*2](#).
- [<Qn>](#) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as [<Qn>*2](#).
- [<Qm>](#) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as [<Qm>*2](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var x : bits(128) = Q(d>>1);
    var y : bits(32) = Q(n>>1)[31:0]; // Note: 32 bits wide
    let w : bits(128) = Q(m>>1);
    for e = 0 to 3 do
        let t : bits(32) = SHAparity(x[63:32], x[95:64], x[127:96]);
        y = y + ROL(x[31:0], 5) + t + w[e*32];
        x[63:32] = ROL(x[63:32], 30);
        let yx : bits(160) = ROL(y::x, 32);
        (y, x) = (yx[128+32], yx[0+128]);
    end;
    Q(d>>1) = x;
end;
```

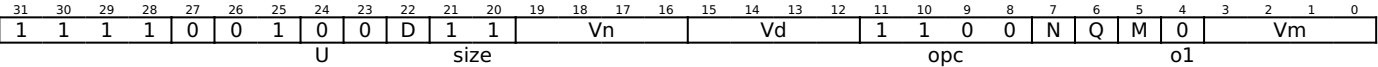
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHA1SU0

SHA1 schedule update 0.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_SHA1)



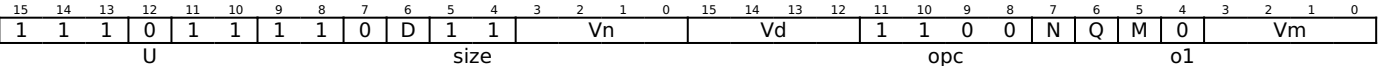
Encoding

SHA1SU0.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_SHA1)



Encoding

SHA1SU0.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let op1 : bits(128) = Q(d>>1);
    var op2 : bits(128) = Q(n>>1);
    let op3 : bits(128) = Q(m>>1);
    op2 = op2[63:0] :: op1[127:64];
    Q(d>>1) = op1 XOR op2 XOR op3;
end;
```

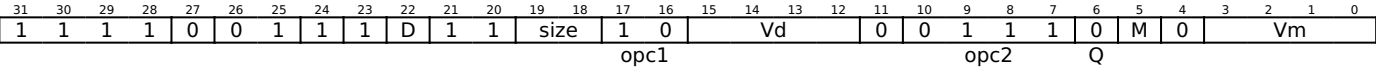
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHA1SU1

SHA1 schedule update 1.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_SHA1)



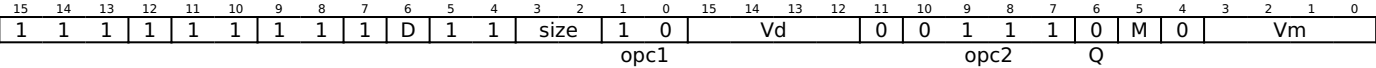
Encoding

SHA1SU1.32 <Qd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_SHA1)



Encoding

SHA1SU1.32 <Qd>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckCryptoEnabled32();
  let X : bits(128) = Q(d>>1);
  let Y : bits(128) = Q(m>>1);
  let T : bits(128) = X XOR LSR(Y, 32);
  let W0 : bits(32) = ROL(T[31:0], 1);
  let W1 : bits(32) = ROL(T[63:32], 1);
  let W2 : bits(32) = ROL(T[95:64], 1);
  let W3 : bits(32) = ROL(T[127:96], 1) XOR ROL(T[31:0], 2);
  Q(d>>1) = W3::W2::W1::W0;
end;
```

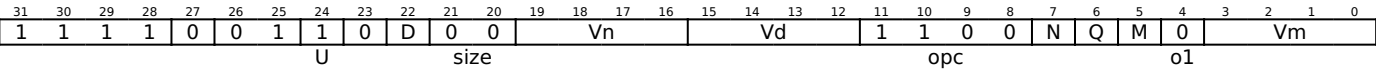
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHA256H

SHA256 hash update part 1.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_SHA256)



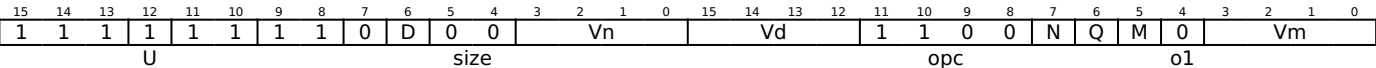
Encoding

SHA256H.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_SHA256)



Encoding

SHA256H.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckCryptoEnabled32();
  let X : bits(128) = Q(d>>1);
  let Y : bits(128) = Q(n>>1);
  let W : bits(128) = Q(m>>1);
  let part1 : boolean = TRUE;
  Q(d>>1) = SHA256hash(X, Y, W, part1);
end;
```

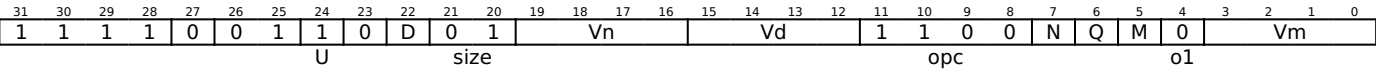
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHA256H2

SHA256 hash update part 2.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_SHA256)



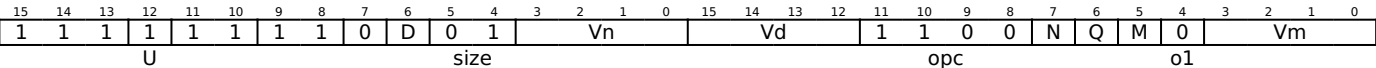
Encoding

SHA256H2.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_SHA256)



Encoding

SHA256H2.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- [<Qd>](#) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as [<Qd>*2](#).
- [<Qn>](#) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as [<Qn>*2](#).
- [<Qm>](#) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as [<Qm>*2](#).

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckCryptoEnabled32();
  let X : bits(128) = Q(n>>1);
  let Y : bits(128) = Q(d>>1);
  let W : bits(128) = Q(m>>1);
  let part1 : boolean = FALSE;
  Q(d>>1) = SHA256hash(X, Y, W, part1);
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHA256SU0

SHA256 schedule update 0.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_SHA256)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0		0	1	1	1	1	M	0	Vm			
opc1																opc2				Q												

Encoding

SHA256SU0.32 <Qd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_SHA256)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd				0	0	1	1	1	1	M	0	Vm				
opc1																opc2				Q											

Encoding

SHA256SU0.32 <Qd>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  var result : bits(128);
  EncodingSpecificOperations();
  CheckCryptoEnabled32();
  let x : bits(128) = Q(d>>1);
  let y : bits(128) = Q(m>>1);
  let t : bits(128) = y[31:0] :: x[127:32];
  for e = 0 to 3 do
    var elt : bits(32) = t[e*32];
    elt = ROR(elt, 7) XOR ROR(elt, 18) XOR LSR(elt, 3);
    result[e*32] = elt + x[e*32];
  end;
  Q(d>>1) = result;
end;
```

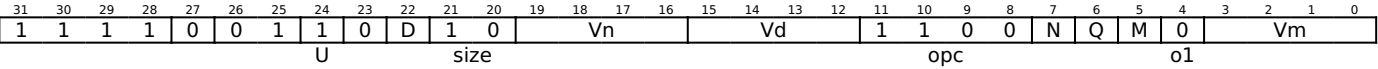
Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

SHA256SU1

SHA256 schedule update 1.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_SHA256)



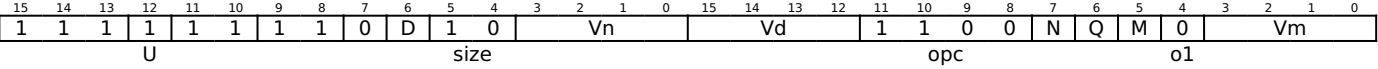
Encoding

SHA256SU1.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_SHA256)



Encoding

SHA256SU1.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var elt : bits(32);
    var result : bits(128);
    let x : bits(128) = Q(d>>1);
    let y : bits(128) = Q(n>>1);
    let z : bits(128) = Q(m>>1);
    let T0 : bits(128) = z[31:0] :: y[127:32];

    var T1 : bits(64) = z[127:64];
    for e = 0 to 1 do
        elt = T1[e*:32];
        elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
        elt = elt + x[e*:32] + T0[e*:32];
        result[e*:32] = elt;
    end;

    T1 = result[63:0];
    for e = 2 to 3 do
        elt = T1[(e - 2)*:32];
        elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
        elt = elt + x[e*:32] + T0[e*:32];
        result[e*:32] = elt;
    end;

    Q(d>>1) = result;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VABA

Vector Absolute Difference and Accumulate subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	1	0	0	1	U	0	D	size				Vn				Vd				0				1	1	1	N	Q	M	1	Vm			
opc																										ol										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	0	D	size				Vn				Vd				0	1	1	1	N	Q	M	1	Vm			
opc																										ol							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c>
For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
This encoding must be unconditional.
For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <q>
See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the operands, encoded in “(U :: size)”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  for r = 0 to regs-1 do
    for e = 0 to elements-1 do
      let op1elt : bits(esize) = Din(n+r)[e*:esize];
      let op2elt : bits(esize) = Din(m+r)[e*:esize];
      let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
      let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
      let absdiff : integer = Abs(element1 - element2);
      if long_destination then
        Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + absdiff;
      else
        D(d+r)[e*:esize] = Din(d+r)[e*:esize] + absdiff;
      end;
    end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VABAL

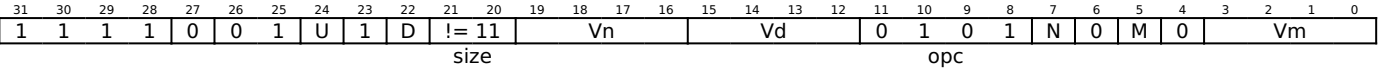
Vector Absolute Difference and Accumulate Long subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



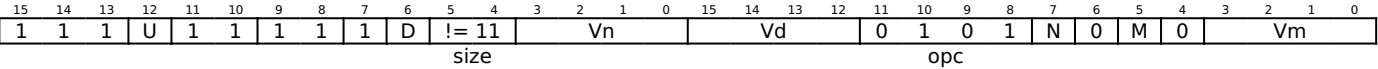
Encoding

VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

T1



Encoding

VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <dt>

Is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let absdiff : integer = Abs(element1 - element2);
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + absdiff;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + absdiff;
            end;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VABD (floating-point)

Vector Absolute Difference (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are floating-point numbers of the same size.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
U												opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
U												opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

- If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.						
	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            D(d+r)[e*:esize] = FPAbs(esize)(FPSub(esize)(op1elt, op2elt, fpcr), fpcr);
        end;
    end;
end;
```

VABD (integer)

Vector Absolute Difference (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	U	0	D	size			Vn				Vd				0				1	1	1	N	Q	M	0	Vm			
											opc										o1														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	0	D	size				Vn				Vd				0	1	1	1	N	Q	M	0	Vm			
											opc										o1												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the operands, encoded in “(U :: size)”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let absdiff : integer = Abs(element1 - element2);
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = absdiff[2*esize-1:0];
            else
                D(d+r)[e*:esize] = absdiff[esize-1:0];
            end;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VABDL (integer)

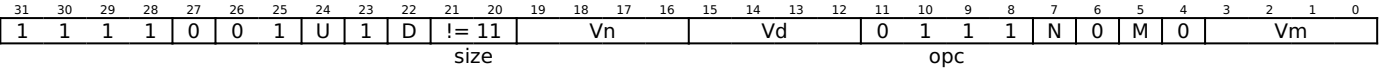
Vector Absolute Difference Long (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



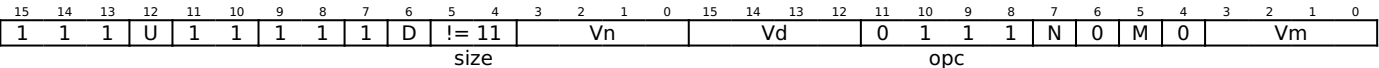
Encoding

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

T1



Encoding

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <dt>

Is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let absdiff : integer = Abs(element1 - element2);
            if long_destination then
                Q(d>>1)[e*(2*esize)] = absdiff[2*esize-1:0];
            else
                D(d+r)[e*:esize] = absdiff[esize-1:0];
            end;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VABS

Vector Absolute takes the absolute value of each element in a vector, and places the results in a second vector. The floating-point version only clears the sign bit.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd			0		F	1	1	0	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABS{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABS{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	0	0	0	Vd			1		0	size	1	1	M	0	Vm				
cond				o1								opc2				o3															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VABS{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VABS{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			0	1	Vd				0	F	1	1	0	Q	M	0	Vm			
opc1																																

Encoding for the 64-bit SIMD vector variant

Applies when `(Q == 0)`

VABS{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when `(Q == 1)`

VABS{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0		Vd			1	0	size	1	1	M	0		Vm			
o1										opc2						o3															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when `(size == 01)`

VABS{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

VABS{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see [Standard assembler syntax fields](#).

For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<dt> Is the data type for the elements of the vectors, encoded in "(F :: size)":

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                if floating_point then
                    D(d+r)[e*:esize] = FPAbs{esize}(D(m+r)[e*:esize], fpcr);
                else
                    let result : integer = Abs(SInt(D(m+r)[e*:esize]));
                    D(d+r)[e*:esize] = result[esize-1:0];
                end;
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 => H(d) = FPAbs{16}(H(m), fpcr);
            when 32 => S(d) = FPAbs{32}(S(m), fpcr);
            when 64 => D(d) = FPAbs{64}(D(m), fpcr);
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VACGE

Vector Absolute Compare Greater Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VACLE](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	1	Vm			
U										op		opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	1	Vm			
U										op		opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.						
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .						
<dt>	See Standard assembler syntax fields .						
	Is the data type for the elements of the vectors, encoded in "sz":						
<table> <tr> <th>sz</th><th><dt></th></tr> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </table>		sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = FPAbs{}(D(n+r)[e*:esize], fpcr);
            let op2 : bits(esize) = FPAbs{}(D(m+r)[e*:esize], fpcr);
            var test_passed : boolean;
            if or_equal then
                test_passed = FPCCompareGE{esize}(op1, op2, fpcr);
            else
                test_passed = FPCCompareGT{esize}(op1, op2, fpcr);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VACGT

Vector Absolute Compare Greater Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VACLT](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn			Vd			1	1	1	0	N	Q	M	1	Vm					
U										op		opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn			Vd			1	1	1	0	N	Q	M	1	Vm					
U										op		opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.						
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .						
<dt>	See Standard assembler syntax fields .						
	Is the data type for the elements of the vectors, encoded in "sz":						
<table> <tr> <th>sz</th><th><dt></th></tr> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </table>		sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = FPAbs{}(D(n+r)[e*:esize], fpcr);
            let op2 : bits(esize) = FPAbs{}(D(m+r)[e*:esize], fpcr);
            var test_passed : boolean;
            if or_equal then
                test_passed = FPCCompareGE{esize}(op1, op2, fpcr);
            else
                test_passed = FPCCompareGT{esize}(op1, op2, fpcr);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VACLE

Vector Absolute Compare Less Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of [VACGE](#). This means:

- The encodings in this description are named to match the encodings of [VACGE](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VACGE](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	1	Vm			
U							op					opc							o1												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	1	Vm			
U							op					opc							o1												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

Assembler Symbols

- <c>
- For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <q>
- See *Standard assembler syntax fields*.

<dt>	Is the data type for the elements of the vectors, encoded in “sz”:						
	<table> <tr> <th>sz</th><th><dt></th></tr> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						

Operation

The description of [VACGE](#) gives the operational pseudocode for this instruction.

VACLT

Vector Absolute Compare Less Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of [VACGT](#). This means:

- The encodings in this description are named to match the encodings of [VACGT](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VACGT](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	0	1	1	0	D	1	sz	Vn				Vd				1				1	1	0	N	Q	M	1	Vm			
U							op					opc								o1														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	0	N	Q	M	1	Vm			
U									op				opc								o1										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

Assembler Symbols

- <c>
- For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>
- For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<dt>	Is the data type for the elements of the vectors, encoded in “sz”:						
	<table> <tr> <th>sz</th><th><dt></th></tr> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						

Operation

The description of [VACGT](#) gives the operational pseudocode for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VADD (floating-point)

Vector Add (floating-point) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
U												opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	1	1	Vn				Vd				1	0	size	N	0	M	0	Vm				
cond				o0				o1												o2											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VADD{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VADD{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VADD{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
U												opc												o1							

Encoding for the 64-bit SIMD vector variant

Applies when (`Q == 0`)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (`Q == 1`)

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	1	1	0	0	D	1	1	Vn					Vd					1	0	size		N	0	M	0	Vm			
o0								o1																o2									

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (`size == 01`)

VADD{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (`size == 10`)

VADD{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (`size == 11`)

VADD{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see Standard assembler syntax fields . For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table border="1"> <thead> <tr> <th>sz</th><th><dt></th></tr> </thead> <tbody> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </tbody> </table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                D(d+r)[e*:esize] = FPAdd(esize)(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                H(d) = FPAdd{16}(H(n), H(m), fpcr);
            when 32 =>
                S(d) = FPAdd{32}(S(n), S(m), fpcr);
            when 64 =>
                D(d) = FPAdd{64}(D(n), D(m), fpcr);
        end;
    end;
end;
```


VADD (integer)

Vector Add (integer) adds corresponding elements in two vectors, and places the results in the destination vector. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	0	0	D	size	Vn					Vd					1	0	0	0	N	Q	M	0	Vm				
U										opc										o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	1	1	1	0	D	size	Vn					Vd					1	0	0	0	N	Q	M	0	Vm				
U										opc										o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <dt> See *Standard assembler syntax fields*.
Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = D(n+r)[e*:esize] + D(m+r)[e*:esize];
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VADDHN

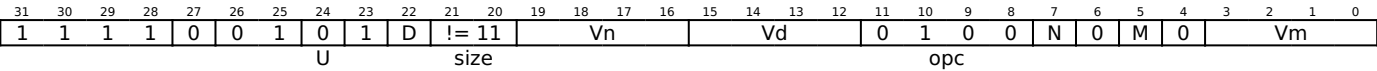
Vector Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are truncated. For rounded results, see [VRADDHN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



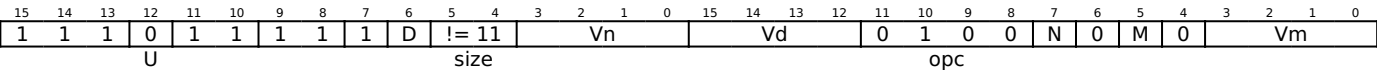
Encoding

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1



Encoding

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1" variant: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the operands, encoded in “size”:
- | size | <dt> |
|------|------|
| 00 | I16 |
| 01 | I32 |
| 10 | I64 |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let result : bits(2*esize) = Qin(n>>1)[e*:(2*esize)] + Qin(m>>1)[e*:(2*esize)];
        D(d)[e*:esize] = result[2*esize-1:esize];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

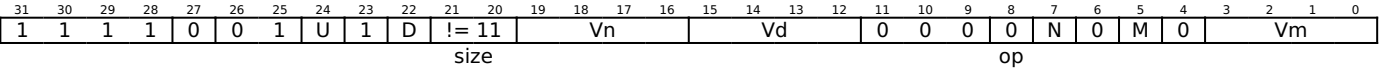
VADDL

Vector Add Long adds corresponding elements in two doubleword vectors, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



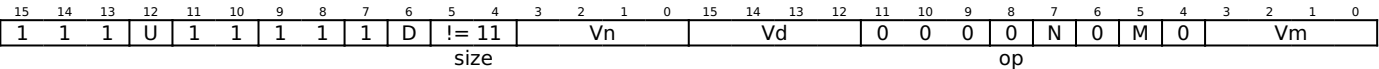
Encoding

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1



Encoding

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <dt>

Is the data type for the elements of the second operand vector, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vaddw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 + element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VADDW

Vector Add Wide adds corresponding elements in one quadword and one doubleword vector, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	!= 11			Vn			Vd			0	0	0	1	N	0	M	0	Vm				
size											op																				

Encoding

VADDW{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	!= 11			Vn			Vd			0	0	0	1	N	0	M	0	Vm				
size											op																				

Encoding

VADDW{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>

For the "T1" variant: see *Standard assembler syntax fields*.
- <dt>

See *Standard assembler syntax fields*.
- Is the data type for the elements of the second operand vector, encoded in "(U :: size)":
- | U | size | <dt> |
|---|------|------|
| 0 | 00 | S8 |
| 0 | 01 | S16 |
| 0 | 10 | S32 |
| 1 | 00 | U8 |
| 1 | 01 | U16 |
| 1 | 10 | U32 |
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vaddw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 + element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VAND (immediate)

Vector Bitwise AND (immediate) performs a bitwise AND between a register value and an immediate value, and returns the result into the destination vector.

This is a pseudo-instruction of [VBIC \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [VBIC \(immediate\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VBIC \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd			0			x	x	1	0	Q	1	1	imm4		
											cmode											op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VAND{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd					1	0	x	1	0	Q	1	1	imm4		
											cmode					op															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VAND{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd					0	x	x	1	0	Q	1	1	imm4							
											cmode											op														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VAND{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			1			0	x	1	0	Q	1	1	imm4		
																cmode				op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VAND{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions*.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Operation

The description of [VBIC \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VBIC \(immediate\)](#) gives the operational information for this instruction.

VAND (register)

Vector Bitwise AND (register) performs a bitwise AND operation between two registers, and places the result in the destination register. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	0	1	0	0	D	0	0	Vn				Vd				0				0	0	1	N	Q	M	1	Vm			
U							size					opc										o1												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VAND{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			
U										size		opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VAND{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c>
- For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <q>
- See *Standard assembler syntax fields*.
- <dt>
- An optional data type. It is ignored by assemblers, and does not affect the encoding.
- <Dd>
- Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn>
- Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>
- Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>
- Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn>
- Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>
- Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled\(\);
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) AND D(m+r);
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VBIC (immediate)

Vector Bitwise Bit Clear (immediate) performs a bitwise AND between a register value and the complement of an immediate value, and returns the result into the destination vector.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction [VAND \(immediate\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd					0	x	x	1	0	Q	1	1	imm4		
											cmode					op															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIC{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIC{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd					1			0	x	1	0	Q	1	1	imm4		
											cmode											op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIC{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIC{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd					0	x	x	1	0	Q	1	1	imm4							
											cmode											op														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIC{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIC{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			1			0			x	1	0	Q	1	1	imm4		
																cmode								op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIC{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIC{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
 - <q> For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
 - <Dd> See [Standard assembler syntax fields](#).
 - <imm> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
 - <Qd> Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 Advanced SIMD instructions](#).
 - <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(d+r) AND NOT(imm64);
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

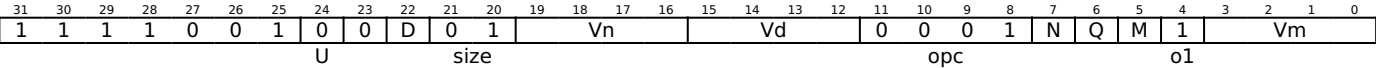
VBIC (register)

Vector Bitwise Bit Clear (register) performs a bitwise AND between a register value and the complement of a register value, and places the result in the destination register.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

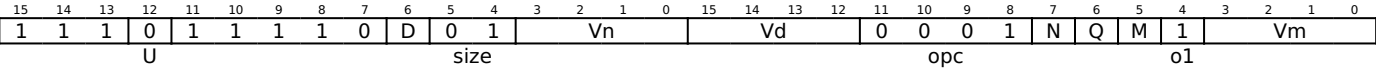
Applies when (Q == 1)

```
VBIC{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VBIC{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled\(\);
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) AND NOT(D(m+r));
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VBIF

Vector Bitwise Insert if False inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	1	Vn			Vd			0			0	0	1	N	Q	M	1	Vm			
U							op				opc										o1										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIF{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIF{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps.VBSL; end;
if op == '10' then operation = VBitOps.VBIT; end;
if op == '11' then operation = VBitOps.VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	1	0	D	1	1	Vn					Vd					0	0	0	1	N	Q	M	1	Vm				
U									op					opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIF{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIF{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps.VBSL; end;
if op == '10' then operation = VBitOps.VBIT; end;
if op == '11' then operation = VBitOps.VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<dt>	See Standard assembler syntax fields .
	An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        case operation of
            when VBitOps VBIF => D(d+r) = (D(d+r) AND D(m+r)) OR (D(n+r) AND NOT(D(m+r)));
            when VBitOps VBIT => D(d+r) = (D(n+r) AND D(m+r)) OR (D(d+r) AND NOT(D(m+r)));
            when VBitOps VBSL => D(d+r) = (D(n+r) AND D(d+r)) OR (D(m+r) AND NOT(D(d+r)));
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

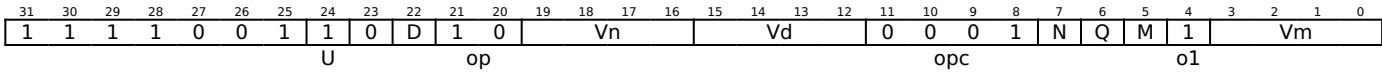
VBIT

Vector Bitwise Insert if True inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIT{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

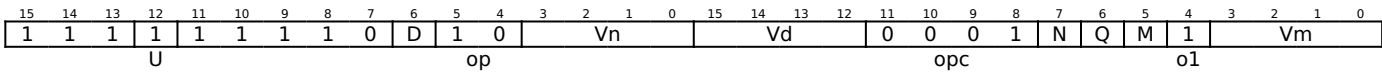
Applies when (Q == 1)

VBIT{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps VBSL; end;
if op == '10' then operation = VBitOps VBIT; end;
if op == '11' then operation = VBitOps VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIT{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIT{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps VBSL; end;
if op == '10' then operation = VBitOps VBIT; end;
if op == '11' then operation = VBitOps VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c>
- For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>
- For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <dt>
- See *Standard assembler syntax fields*.
-
- An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        case operation of
            when VBitOps VBIF =>  $\underline{D}(d+r) = (\underline{D}(d+r) \text{ AND } \underline{D}(m+r)) \text{ OR } (\underline{D}(n+r) \text{ AND NOT } (\underline{D}(m+r)))$ ;
            when VBitOps VBIT =>  $\underline{D}(d+r) = (\underline{D}(n+r) \text{ AND } \underline{D}(m+r)) \text{ OR } (\underline{D}(d+r) \text{ AND NOT } (\underline{D}(m+r)))$ ;
            when VBitOps VBSL =>  $\underline{D}(d+r) = (\underline{D}(n+r) \text{ AND } \underline{D}(d+r)) \text{ OR } (\underline{D}(m+r) \text{ AND NOT } (\underline{D}(d+r)))$ ;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VBSL

Vector Bitwise Select sets each bit in the destination to the corresponding bit from the first source operand when the original destination bit was 1, otherwise from the second source operand.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			
U							op				opc															o1					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBSL{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBSL{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps.VBSL; end;
if op == '10' then operation = VBitOps.VBIT; end;
if op == '11' then operation = VBitOps.VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			
U									op				opc															o1			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBSL{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBSL{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps.VBSL; end;
if op == '10' then operation = VBitOps.VBIT; end;
if op == '11' then operation = VBitOps.VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional.
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> .
<dt>	See <i>Standard assembler syntax fields</i> .
	An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        case operation of
            when VBitOps VBIF => D(d+r) = (D(d+r) AND D(m+r)) OR (D(n+r) AND NOT(D(m+r)));
            when VBitOps VBIT => D(d+r) = (D(n+r) AND D(m+r)) OR (D(d+r) AND NOT(D(m+r)));
            when VBitOps VBSL => D(d+r) = (D(n+r) AND D(d+r)) OR (D(m+r) AND NOT(D(d+r)));
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCADD

Vector Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_FCMA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	rot	1	D	0	S	Vn				Vd				1	0	0	0	N	Q	M	0	Vm			
											op3			op4			U														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCADD{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if S == '0' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 16 << UInt(S);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

(FEAT_FCMA)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	rot	1	D	0	S	Vn			Vd			1	0	0	0	N	Q	M	0	Vm					
											op3			op4			U														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCADD{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if S == '0' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 16 << UInt(S);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```


Assembler Symbols

- <q>

See *Standard assembler syntax fields*.
- <dt>

Is the data type for the elements of the vectors, encoded in “S”:

S	<dt>
0	F16
1	F32
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <rotate>

Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in “rot”:

rot	<rotate>
0	90
1	270
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    let operand1 : bits(64) = D(n+r);
    let operand2 : bits(64) = D(m+r);
    let fpcr : FPCR_Type = StandardFPCR();
    for e = 0 to (elements DIV 2)-1 do
        var element1 : bits(esize);
        var element3 : bits(esize);
        case rot of
            when '0' =>
                element1 = FPNeg{esize}(operand2[(e*2+1)*:esize], fpcr);
                element3 = operand2[(e*2)*:esize];
            when '1' =>
                element1 = operand2[(e*2+1)*:esize];
                element3 = FPNeg{esize}(operand2[(e*2)*:esize], fpcr);
        end;
        let result1 : bits(esize) = FPAdd{}(operand1[(e*2)*:esize], element1, fpcr);
        let result2 : bits(esize) = FPAdd{}(operand1[(e*2+1)*:esize], element3, fpcr);
        D(d+r)[(e*2)*:esize] = result1;
        D(d+r)[(e*2+1)*:esize] = result2;
    end;
end;
```

VCEQ (immediate #0)

Vector Compare Equal to Zero takes each element in a vector, and compares it with zero. If it is equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	1	0	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			0	1	Vd				0	F	0	1	0	Q	M	0	Vm			
opc1																																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<dt>	See Standard assembler syntax fields .
	Is the data type for the elements of the operands, encoded in "(F :: size)":

F	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{'0'};
                test_passed = FPCCompareEQ(esize)(D(m+r)[e*:esize], zero, StandardFPCR());
            else
                test_passed = (D(m+r)[e*:esize] == Zeros(esize));
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCEQ (register)

Vector Compare Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size	Vn				Vd				1	0	0	0	N	Q	M	1	Vm				
U										opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let int_operation : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn			Vd			1			1	1	0	N	Q	M	0	Vm			
U										opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let int_operation : boolean = FALSE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	1	1	0	D	size	Vn					Vd					1	0	0	0	N	Q	M	1	Vm				
U										opc										o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

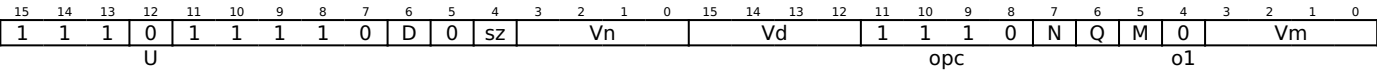
Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let int_operation : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let int_operation : boolean = FALSE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

<q> For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

<dt> See *Standard assembler syntax fields*.

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "T1 128-bit SIMD vector", and "T1 64-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "size":

size	<dt>
00	I8
01	I16
10	I32

For the "A2 128-bit SIMD vector", "A2 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            var test_passed : boolean;
            if int_operation then
                test_passed = (op1elt == op2elt);
            else
                test_passed = FPCompareEQ(esize)(op1elt, op2elt, StandardFPCR());
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;

```

VCGE (immediate #0)

Vector Compare Greater Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is greater than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	1	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size	0	1	Vd	0	F	0	0	1	Q	M	0	Vm								
opc1																																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<dt>	See Standard assembler syntax fields .
	Is the data type for the elements of the operands, encoded in "(F :: size)":

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{'0'};
                test_passed = FPCCompareGE(esize)(D(m+r)[e*:esize], zero, StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) >= 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCGE (register)

Vector Compare Greater Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VCLE \(register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	U	0	D	size	Vn				Vd				0				0	1	1	N	Q	M	1	Vm					
												opc												o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGEType = if U == '1' then VCGEType_unsigned else VCGEType_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			
U												opc												o1							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let vtype : VCGEType = VCGEType_fp;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size	Vn				Vd				0	0	1	1	N	Q	M	1	Vm				
												opc																o1			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

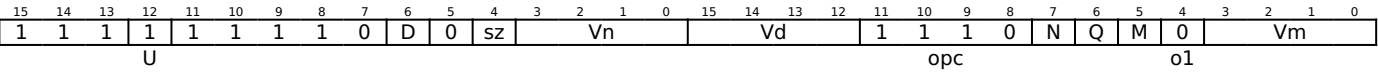
Applies when (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGEType = if U == '1' then VCGEType_unsigned else VCGEType_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let vtype : VCGEType = VCGEType_fp;
let esize : integer{} = 32 >> UInt(sz);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .

<dt> For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "T1 128-bit SIMD vector", and "T1 64-bit SIMD vector" variants: is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the "A2 128-bit SIMD vector", "A2 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  for r = 0 to regs-1 do
    for e = 0 to elements-1 do
      let op1elt : bits(esize) = D(n+r)[e*:esize];
      let op2elt : bits(esize) = D(m+r)[e*:esize];
      var test_passed : boolean;
      case vtype of
        when VCGEType_signed => test_passed = (SInt(op1elt) >= SInt(op2elt));
        when VCGEType_unsigned => test_passed = (UInt(op1elt) >= UInt(op2elt));
        when VCGEType_fp => test_passed = FPCompareGE(esize)(op1elt, op2elt,
                                                                StandardFPCR());
      end;
      D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
    end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VCGT (immediate #0)

Vector Compare Greater Than Zero takes each element in a vector, and compares it with zero. If it is greater than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	0	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	1		Vd				0	F	0	0	0	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<dt>	Is the data type for the elements of the operands, encoded in "(F :: size)":

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{'0'};
                test_passed = FPCCompareGT(esize)(D(m+r)[e*:esize], zero, StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) > 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCGT (register)

Vector Compare Greater Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VCLT \(register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn				Vd				0	0	1	1	N	Q	M	0	Vm				
																opc				o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGTtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	1	sz	Vn			Vd			1			1	1	0	N	Q	M	0	Vm			
																opc				o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let vtype : VCGTtype = VCGTtype_fp;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size	Vn				Vd				0	0	1	1	N	Q	M	0	Vm				
																opc						o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

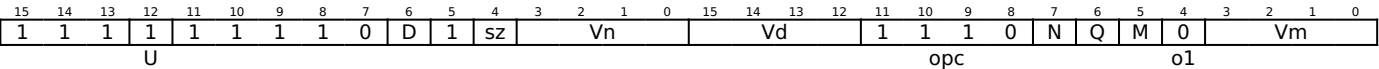
Applies when (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGTtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let vtype : VCGTtype = VCGTtype_fp;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .

<dt> For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "T1 128-bit SIMD vector", and "T1 64-bit SIMD vector" variants: is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the "A2 128-bit SIMD vector", "A2 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  for r = 0 to regs-1 do
    for e = 0 to elements-1 do
      let op1elt : bits(esize) = D(n+r)[e*:esize];
      let op2elt : bits(esize) = D(m+r)[e*:esize];
      var test_passed : boolean;
      case vtype of
        when VCGTtype_signed => test_passed = (SInt(op1elt) > SInt(op2elt));
        when VCGTtype_unsigned => test_passed = (UInt(op1elt) > UInt(op2elt));
        when VCGTtype_fp => test_passed = FPCompareGT(esize)(op1elt, op2elt,
                                                                StandardFPCR());
      end;
      D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
    end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VCLE (immediate #0)

Vector Compare Less Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is less than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd			0		F	0	1	1	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	1	Vd	0	F	0	1	1	Q	M	0	Vm							
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<dt>	Is the data type for the elements of the operands, encoded in "(F :: size)":

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{'0'};
                test_passed = FPCCompareGE(esize)(zero, D(m+r)[e*:esize], StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) <= 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLE (register)

Vector Compare Less Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of [VCGE \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VCGE \(register\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VCGE \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn				Vd				0	0	1	1	N	Q	M	1	Vm				
opc																								o1							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$\text{VCLE}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \{ \langle Dd \rangle, \} \langle Dn \rangle, \langle Dm \rangle$

is equivalent to

$\text{VCGE}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \langle Dd \rangle, \langle Dm \rangle, \langle Dn \rangle$

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

$\text{VCLE}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \{ \langle Qd \rangle, \} \langle Qn \rangle, \langle Qm \rangle$

is equivalent to

$\text{VCGE}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \langle Qd \rangle, \langle Qm \rangle, \langle Qn \rangle$

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn			Vd			1			1	1	0	N	Q	M	0	Vm			
U												opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$\text{VCLE}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \{ \langle Dd \rangle, \} \langle Dn \rangle, \langle Dm \rangle$

is equivalent to

$\text{VCGE}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \langle Dd \rangle, \langle Dm \rangle, \langle Dn \rangle$

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

$\text{VCLE}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \{ \langle Qd \rangle, \} \langle Qn \rangle, \langle Qm \rangle$

is equivalent to

$\text{VCGE}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \langle Qd \rangle, \langle Qm \rangle, \langle Qn \rangle$

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size	Vn				Vd				0	0	1	1	N	Q	M	1	Vm				
																opc								o1							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$\text{VCLE}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \{ \langle Dd \rangle, \} \langle Dn \rangle, \langle Dm \rangle$

is equivalent to

VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

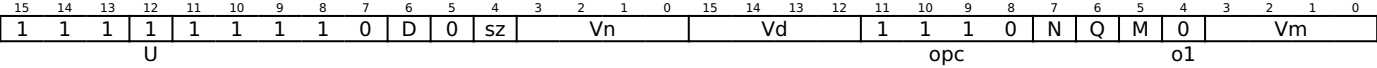
Applies when (Q == 1)

VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <dt>

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "T1 128-bit SIMD vector", and "T1 64-bit SIMD vector" variants: is the data type for the elements of the operands, encoded in "U :: size":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the "A2 128-bit SIMD vector", "A2 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16

- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of [VCGE \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VCGE \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLS

Vector Count Leading Sign Bits determines the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector. The result does not include the topmost bit itself.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit signed integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	1	0	0	1	1	1	D	1	1	size				Vd				0				1	0				0	Q	M	0	Vm			
												opc1																opc2								

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0			Vd			0	1	0	0	0	Q	M	0			Vm	
												opc1																opc2			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c>
- For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <q>
- See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = CountLeadingSignBits(D(m+r)[e*:esize])[esize-1:0];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLT (immediate #0)

Vector Compare Less Than Zero takes each element in a vector, and compares it with zero. If it is less than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	1	0	0	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	1	Vd				0	F	1	0	0	Q	M	0	Vm				
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && size == '00' then Undefined(); end;
if F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<dt>	Is the data type for the elements of the operands, encoded in "(F :: size)":

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{}('0');
                test_passed = FPCCompareGT(esize){zero, D(m+r)[e*:esize], StandardFPCR()};
            else
                test_passed = (SInt(D(m+r)[e*:esize]) < 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLT (register)

Vector Compare Less Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of [VCGT \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VCGT \(register\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VCGT \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn				Vd				0	0	1	1	N	Q	M	0	Vm				
												opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$VCLT\{<c>\}\{<q>\}.<dt>\{<Dd>, \}<Dn>, <Dm>$

is equivalent to

$VCGT\{<c>\}\{<q>\}.<dt> <Dd>, <Dm>, <Dn>$

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

$VCLT\{<c>\}\{<q>\}.<dt>\{<Qd>, \}<Qn>, <Qm>$

is equivalent to

$VCGT\{<c>\}\{<q>\}.<dt> <Qd>, <Qm>, <Qn>$

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn			Vd			1			1	1	0	N	Q	M	0	Vm			
U												opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$VCLT\{<c>\}\{<q>\}.<dt>\{<Dd>, \}<Dn>, <Dm>$

is equivalent to

$VCGT\{<c>\}\{<q>\}.<dt> <Dd>, <Dm>, <Dn>$

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

$VCLT\{<c>\}\{<q>\}.<dt>\{<Qd>, \}<Qn>, <Qm>$

is equivalent to

$VCGT\{<c>\}\{<q>\}.<dt> <Qd>, <Qm>, <Qn>$

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size	Vn				Vd				0	0	1	1	N	Q	M	0	Vm				
																opc										o1					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$VCLT\{<c>\}\{<q>\}.<dt>\{<Dd>, \}<Dn>, <Dm>$

is equivalent to

$VCGT\{<c>\}\{<q>\}.<dt> <Dd>, <Dm>, <Dn>$

Encoding for the 128-bit SIMD vector variant

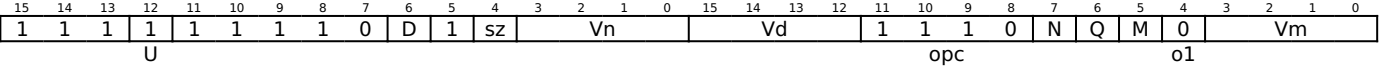
Applies when (Q == 1)

$VCLT\{<c>\}\{<q>\}.<dt> \{<Qd>, \}<Qn>, <Qm>$

is equivalent to

$VCGT\{<c>\}\{<q>\}.<dt> <Qd>, <Qm>, <Qn>$

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$VCLT\{<c>\}\{<q>\}.<dt> \{<Dd>, \}<Dn>, <Dm>$

is equivalent to

$VCGT\{<c>\}\{<q>\}.<dt> <Dd>, <Dm>, <Dn>$

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

$VCLT\{<c>\}\{<q>\}.<dt> \{<Qd>, \}<Qn>, <Qm>$

is equivalent to

$VCGT\{<c>\}\{<q>\}.<dt> <Qd>, <Qm>, <Qn>$

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <dt>

See *Standard assembler syntax fields*.
- For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "T1 128-bit SIMD vector", and "T1 64-bit SIMD vector" variants: is the data type for the elements of the operands, encoded in “(U :: size)”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the "A2 128-bit SIMD vector", "A2 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of [VCGT \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VCGT \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers. The result vector elements are the same data type as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size		opc1		Vd				0		1	0		0	1	Q	M	0	Vm		
												opc1																opc2				

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	D	1	1	size	0	0			Vd			0	1	0	0	1	Q	M	0			Vm
												opc1																opc2			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional.
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . See <i>Standard assembler syntax fields</i> .

<dt>	Is the data type for the elements of the operands, encoded in “size”:								
	<table> <tr> <th>size</th><th><dt></th></tr> <tr> <td>00</td><td>I8</td></tr> <tr> <td>01</td><td>I16</td></tr> <tr> <td>10</td><td>I32</td></tr> </table>	size	<dt>	00	I8	01	I16	10	I32
size	<dt>								
00	I8								
01	I16								
10	I32								
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.								
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.								
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.								
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.								

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = CountLeadingZeroBits(D(m+r)[e*:esize])[esize-1:0];
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCMLA

Vector Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
 - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
 - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_FCMA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	rot		D	1	S	Vn				Vd				1	0	0	0	N	Q	M	0	Vm			
											op3			op4			U														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then Undefined(); end;
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

(FEAT_FCMA)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	rot		D	1	S	Vn				Vd				1	0	0	0	N	Q	M	0	Vm			
											op3			op4			U														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then Undefined(); end;
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <q>

See *Standard assembler syntax fields*.
- <dt>

Is the data type for the elements of the vectors, encoded in “S”:

S	<dt>
0	F16
1	F32
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <rotate>

Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  let operand1 : bits(64) = D(n+r);
  let operand2 : bits(64) = D(m+r);
  let operand3 : bits(64) = D(d+r);
  for e = 0 to (elements DIV 2)-1 do
    var element1 : bits(esize);
    var element2 : bits(esize);
    var element3 : bits(esize);
    var element4 : bits(esize);
    case rot of
      when '00' =>
        element1 = operand2[(e*2)*:esize];
        element2 = operand1[(e*2)*:esize];
        element3 = operand2[(e*2+1)*:esize];
        element4 = operand1[(e*2)*:esize];
      when '01' =>
        element1 = FPNeg{esize}(operand2[(e*2+1)*:esize], fpcr);
        element2 = operand1[(e*2+1)*:esize];
        element3 = operand2[(e*2)*:esize];
        element4 = operand1[(e*2+1)*:esize];
      when '10' =>
        element1 = FPNeg{esize}(operand2[(e*2)*:esize], fpcr);
        element2 = operand1[(e*2)*:esize];
        element3 = FPNeg{esize}(operand2[(e*2+1)*:esize], fpcr);
        element4 = operand1[(e*2)*:esize];
      when '11' =>
        element1 = operand2[(e*2+1)*:esize];
        element2 = operand1[(e*2+1)*:esize];
        element3 = FPNeg{esize}(operand2[(e*2)*:esize], fpcr);
        element4 = operand1[(e*2+1)*:esize];
    end;
    let result1 : bits(esize) = FPMulAdd{(operand3[(e*2)*:esize],
                                         element2,element1, fpcr);
    let result2 : bits(esize) = FPMulAdd{(operand3[(e*2+1)*:esize],
                                         element4,element3,fpcr);
    D(d+r)[(e*2)*:esize] = result1;
    D(d+r)[(e*2+1)*:esize] = result2;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCMLA (by element)

Vector Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
 - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
 - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_FCMA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	S	D	rot	Vn				Vd				1	0	0	0	N	Q	M	0	Vm				
																												U			

Encoding for the 64-bit SIMD vector of half-precision floating-point variant

Applies when (S == 0 && Q == 0)

VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>

Encoding for the 64-bit SIMD vector of single-precision floating-point variant

Applies when (S == 1 && Q == 0)

VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>

Encoding for the 128-bit SIMD vector of half-precision floating-point variant

Applies when (S == 0 && Q == 1)

VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>

Encoding for the 128-bit SIMD vector of single-precision floating-point variant

Applies when (S == 1 && Q == 1)

VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if S == '1' then UInt(M::Vm) else UInt(Vm);
let index : integer = if S == '1' then 0 else UInt(M);
let esize : integer{} = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then Undefined(); end;
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

(FEAT_FCMA)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	S	D	rot	Vn				Vd				1	0	0	0	N	Q	M	0	Vm				
U																															

Encoding for the 64-bit SIMD vector of half-precision floating-point variant

Applies when (S == 0 && Q == 0)

VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>

Encoding for the 64-bit SIMD vector of single-precision floating-point variant

Applies when (S == 1 && Q == 0)

VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>

Encoding for the 128-bit SIMD vector of half-precision floating-point variant

Applies when (S == 0 && Q == 1)

VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>

Encoding for the 128-bit SIMD vector of single-precision floating-point variant

Applies when (S == 1 && Q == 1)

VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FCMA) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if S == '1' then UInt(M::Vm) else UInt(Vm);
let index : integer = if S == '1' then 0 else UInt(M);
let esize : integer{} = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then Undefined(); end;
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .										
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.										
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.										
<Dm>	For the "A1 128-bit SIMD vector of half-precision floating-point", "A1 64-bit SIMD vector of half-precision floating-point", "T1 128-bit SIMD vector of half-precision floating-point", and "T1 64-bit SIMD vector of half-precision floating-point" variants: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field. For the "A1 128-bit SIMD vector of single-precision floating-point", "A1 64-bit SIMD vector of single-precision floating-point", "T1 128-bit SIMD vector of single-precision floating-point", and "T1 64-bit SIMD vector of single-precision floating-point" variants: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.										
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.										
<rotate>	Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in "rot": <table><tr><th>rot</th><th><rotate></th></tr><tr><td>00</td><td>0</td></tr><tr><td>01</td><td>90</td></tr><tr><td>10</td><td>180</td></tr><tr><td>11</td><td>270</td></tr></table>	rot	<rotate>	00	0	01	90	10	180	11	270
rot	<rotate>										
00	0										
01	90										
10	180										
11	270										
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.										
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.										

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  let operand1 : bits(64) = D(n+r);
  let operand2 : bits(64) = Din(m);
  let operand3 : bits(64) = D(d+r);
  for e = 0 to (elements DIV 2)-1 do
    var element1 : bits(esize);
    var element2 : bits(esize);
    var element3 : bits(esize);
    var element4 : bits(esize);
    case rot of
      when '00' =>
        element1 = operand2[(index*2)*:esize];
        element2 = operand1[(e*2)*:esize];
        element3 = operand2[(index*2+1)*:esize];
        element4 = operand1[(e*2)*:esize];
      when '01' =>
        element1 = FPNeg{esize}(operand2[(index*2+1)*:esize], fpcr);
        element2 = operand1[(e*2+1)*:esize];
        element3 = operand2[(index*2)*:esize];
        element4 = operand1[(e*2+1)*:esize];
      when '10' =>
        element1 = FPNeg{esize}(operand2[(index*2)*:esize], fpcr);
        element2 = operand1[(e*2)*:esize];
        element3 = FPNeg{esize}(operand2[(index*2+1)*:esize], fpcr);
        element4 = operand1[(e*2)*:esize];
      when '11' =>
        element1 = operand2[(index*2+1)*:esize];
        element2 = operand1[(e*2+1)*:esize];
        element3 = FPNeg{esize}(operand2[(index*2)*:esize], fpcr);
        element4 = operand1[(e*2+1)*:esize];
    end;
    let result1 : bits(esize) = FPMulAdd{}(operand3[(e*2)*:esize],
                                           element2,element1, fpcr);
    let result2 : bits(esize) = FPMulAdd{}(operand3[(e*2+1)*:esize],
                                           element4,element3,fpcr);
    D(d+r)[(e*2)*:esize] = result1;
    D(d+r)[(e*2+1)*:esize] = result2;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCMP

Vector Compare compares two floating-point registers, or one floating-point register and zero. It writes the result to the **FPSCR** flags. These are normally transferred to the **PSTATE**. {N, Z, C, V} Condition flags by a subsequent **VMRS** instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	1	0	0	Vd				1	0	size		0	1	M	0	Vm			
cond				o1								opc2				E															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCMP{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as **NOP**. This means it behaves as if it fails the Condition code check.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				1	1	1	0	1	D	1	1	0	1	0	1	Vd				1	0	size		0	1	(0)	0	(0)	(0)	(0)	(0)				
cond				o1								opc2				E												M				Vm			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCMP{<c>}{<q>}.F16 <Sd>, #0.0

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, #0.0

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, #0.0

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd				1	0	size		0	1	M	0	Vm			
										o1		opc2															E				

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCMP{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd				1	0	size		0	1	(0)	0	(0)	(0)	(0)	(0)				
										o1		opc2															E				M	Vm			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCMP{<c>}{<q>}.F16 <Sd>, #0.0

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    var nzcvcv : bits(4);
    case esize of
        when 16 =>
            let op16 : bits(16) = if with_zero then FPZero{16}('0') else H(m);
            nzcvcv = FPCompare{16}(H(d), op16, quiet_nan_exc, fpcr);
        when 32 =>
            let op32 : bits(32) = if with_zero then FPZero{32}('0') else S(m);
            nzcvcv = FPCompare{32}(S(d), op32, quiet_nan_exc, fpcr);
        when 64 =>
            let op64 : bits(64) = if with_zero then FPZero{64}('0') else D(m);
            nzcvcv = FPCompare{64}(D(d), op64, quiet_nan_exc, fpcr);
    end;

    FPSCR()[31:28] = nzcvcv; // FPSCR.[N,Z,C,V] set to nzcvcv
end;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the [FPSCR](#) condition flags to N=0, Z=0, C=1, and V=1.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCMPE

Vector Compare, raising Invalid Operation on NaN compares two floating-point registers, or one floating-point register and zero. It writes the result to the *FPSCR* flags. These are normally transferred to the *PSTATE*.{N, Z, C, V} Condition flags by a subsequent *VMRS* instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be *UNDEFINED*, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	1	0	0	Vd				1	0	size		1	1	M	0	Vm			
cond												o1		opc2								E									

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is *UNDEFINED*.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	1	0	1	Vd				1	0	size		1	1	(0)	0	(0)	(0)	(0)	(0)
cond												o1		opc2								E		M		Vm					

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCMPE{<c>}{<q>}.F16 <Sd>, #0.0

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, #0.0

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMPPE{<c>}{<q>}.F64 <Dd>, #0.0

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd			1	0	size		1	1	M	0	Vm				
										o1		opc2														E					

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCMPPE{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMPPE{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMPPE{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd			1			0	size		1	1	(0)	0	(0)	(0)	(0)	(0)
										o1		opc2														E		M		Vm		

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCMPPE{<c>}{<q>}.F16 <Sd>, #0.0

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    var nzcvcv : bits(4);
    case esize of
        when 16 =>
            let op16 : bits(16) = if with_zero then FPZero{16}('0') else H(m);
            nzcvcv = FPCompare{16}(H(d), op16, quiet_nan_exc, fpcr);
        when 32 =>
            let op32 : bits(32) = if with_zero then FPZero{32}('0') else S(m);
            nzcvcv = FPCompare{32}(S(d), op32, quiet_nan_exc, fpcr);
        when 64 =>
            let op64 : bits(64) = if with_zero then FPZero{64}('0') else D(m);
            nzcvcv = FPCompare{64}(D(d), op64, quiet_nan_exc, fpcr);
    end;

    FPSCR()[31:28] = nzcvcv; // FPSCR.[N,Z,C,V] set to nzcvcv
end;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the [FPSCR](#) condition flags to N=0, Z=0, C=1, and V=1.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCNT

Vector Count Set Bits counts the number of bits that are one in each element in a vector, and places the results in a second vector.

The operand vector elements must be 8-bit fields.

The result vector elements are 8-bit integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	0	1	0	Q	M	0	Vm			
opc1																opc2															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCNT{<c>}{<q>}.8 <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCNT{<c>}{<q>}.8 <Qd>, <Qm>

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8;
let elements : integer = 8;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0			Vd		0	1	0	1	0	Q	M	0			Vm		
opc1																opc2															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCNT{<c>}{<q>}.8 <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCNT{<c>}{<q>}.8 <Qd>, <Qm>

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8;
let elements : integer = 8;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<Dd>	See Standard assembler syntax fields .
<Dm>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled\(\);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = BitCount(D(m+r)[e*:esize])[esize-1:0];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between double-precision and single-precision)

Convert between double-precision and single-precision does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	1	x	1	1	M	0	Vm			
cond								o1				opc2								size				o3							

Encoding for the Single-precision to double-precision variant

Applies when (size == 10)

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Encoding for the Double-precision to single-precision variant

Applies when (size == 11)

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let double_to_single : boolean = (size == '11');
let d : integer = if double_to_single then UInt(Vd::D) else UInt(D::Vd);
let m : integer = if double_to_single then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	1	x	1	1	M	0	Vm			
								o1				opc2								size				o3							

Encoding for the Single-precision to double-precision variant

Applies when (size == 10)

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Encoding for the Double-precision to single-precision variant

Applies when (size == 11)

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let double_to_single : boolean = (size == '11');
let d : integer = if double_to_single then UInt(Vd::D) else UInt(D::Vd);
let m : integer = if double_to_single then UInt(M::Vm) else UInt(Vm::M);
```

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    if double to single then
        S(d) = FPConvert{32, 64}(D(m), EffectiveFPCR());
    else
        D(d) = FPConvert{64, 32}(S(m), EffectiveFPCR());
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between floating-point and fixed-point, Advanced SIMD)

Vector Convert between floating-point and fixed-point converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	!= 000xxx						Vd				1	1	op		0	Q	M	1	Vm			
										imm6												L									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if op[1] == '0' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if op[1] == '0' && imm6 == '10xxxx' then Undefined(); end;
if imm6 == '0xxxxx' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let to_fixed : boolean = (op[0] == '1');
let frac_bits : integer = 64 - UInt(imm6);
let unsigned : boolean = (U == '1');
let esize : integer{} = 16 << UInt(op[1]);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	!= 000xxx						Vd				1	1	op	0	Q	M	1	Vm				
										imm6												L									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if op[1] == '0' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if op[1] == '0' && imm6 == '10xxxx' then Undefined(); end;
if imm6 == '0xxxxx' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let to_fixed : boolean = (op[0] == '1');
let frac_bits : integer = 64 - UInt(imm6);
let unsigned : boolean = (U == '1');
let esize : integer{} = 16 << UInt(op[1]);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<dt1> Is the data type for the elements of the destination vector, encoded in “(op :: U)”:

op	U	<dt1>
00	x	F16
01	0	S16
01	1	U16
10	x	F32
11	0	S32
11	1	U32

<dt2> Is the data type for the elements of the source vector, encoded in “(op :: U)”:

op	U	<dt2>
00	0	S16
00	1	U16
01	x	F16
10	0	S32
10	1	U32
11	x	F32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<fbits> The number of fraction bits in the fixed point number, in the range 1 to 32 for 32-bit elements, or in the range 1 to 16 for 16-bit elements:

- (64 - <fbits>) is encoded in imm6.

An assembler can permit an <fbits> value of 0. This is encoded as floating-point to integer or integer to floating-point instruction, see [VCVT \(between floating-point and integer, Advanced SIMD\)](#).

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var result : bits(esize);
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(m+r)[e*:esize];
            if to_fixed then
                result = FPToFixed(esize, esize)(operand1, frac_bits, unsigned, fpcr,
                                                    FPRounding_ZERO);
            else
                result = FixedToFP(esize, esize)(operand1, frac_bits, unsigned, fpcr,
                                                    FPRounding_TIEEVEN);
            end;
            D(d+r)[e*:esize] = result;
        end;
    end;
end;

```


VCVT (between floating-point and fixed-point, floating-point)

Convert between floating-point and fixed-point converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point. Software can specify the fixed-point value as either signed or unsigned.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	1	op	1	U	Vd				1	0	sf		sx	1	i	0	imm4			
cond										o1																					

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (op == 0 && sf == 01)

VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (op == 1 && sf == 01)

VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>

Encoding for the Single-precision scalar variant

Applies when (op == 0 && sf == 10)

VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>

Encoding for the Single-precision scalar variant

Applies when (op == 1 && sf == 10)

VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>

Encoding for the Double-precision scalar variant

Applies when (op == 0 && sf == 11)

VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>

Encoding for the Double-precision scalar variant

Applies when (op == 1 && sf == 11)

VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>

Decode for all variants of this encoding

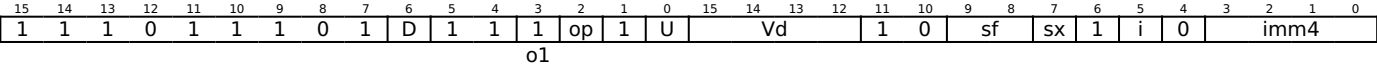
```
if sf == '00' || (sf == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if sf == '01' && cond != '1110' then UnpredictableProcedure(); end;
let to_fixed : boolean = (op == '1');
let unsigned : boolean = (U == '1');
let size : integer{} = if sx == '0' then 16 else 32;
let frac_bits : integer = size - UInt(imm4::i);
let fp_size : integer = 8 << UInt(sf);
let d : integer = if sf == '11' then UInt(D::Vd) else UInt(Vd::D);
if frac_bits < 0 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (op == 0 && sf == 01)

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (op == 1 && sf == 01)

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Single-precision scalar variant

Applies when (op == 0 && sf == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Single-precision scalar variant

Applies when (op == 1 && sf == 10)

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Double-precision scalar variant

Applies when (op == 0 && sf == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Encoding for the Double-precision scalar variant

Applies when (op == 1 && sf == 11)

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

Decode for all variants of this encoding

```
if sf == '00' || (sf == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if sf == '01' && InITBlock() then UnpredictableProcedure(); end;
let to_fixed : boolean = (op == '1');
let unsigned : boolean = (U == '1');
let size : integer{} = if sx == '0' then 16 else 32;
let frac_bits : integer = size - UInt(imm4::i);
let fp_size : integer = 8 << UInt(sf);
let d : integer = if sf == '11' then UInt(D::Vd) else UInt(Vd::D);
if frac_bits < 0 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VCVT (between floating-point and fixed-point)*.

Assembler Symbols

- <c>

See *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <dt>

Is the data type for the fixed-point number, encoded in “(U :: sx)”:

U	sx	<dt>
0	0	S16
0	1	S32
1	0	U16
1	1	U32

<Sdm>

Is the 32-bit name of the SIMD&FP destination and source register, encoded in the "Vd:D" field.

<fbits>

The number of fraction bits in the fixed-point number:

- If <dt> is S16 or U16, <fbits> must be in the range 0-16. (16 - <fbits>) is encoded in [imm4, i]
- If <dt> is S32 or U32, <fbits> must be in the range 1-32. (32 - <fbits>) is encoded in [imm4, i].

<Ddm>

Is the 64-bit name of the SIMD&FP destination and source register, encoded in the "D:Vd" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    if to_fixed then
        var result : bits(size);
        case fp_size of
            when 16 =>
                result = FPToFixed{size, 16}(H(d), frac_bits, unsigned, fpcr, FPRounding_ZERO);
                S(d) = Extend{32}(result, unsigned);
            when 32 =>
                result = FPToFixed{size, 32}(S(d), frac_bits, unsigned, fpcr, FPRounding_ZERO);
                S(d) = Extend{32}(result, unsigned);
            when 64 =>
                result = FPToFixed{size, 64}(D(d), frac_bits, unsigned, fpcr, FPRounding_ZERO);
                D(d) = Extend{64}(result, unsigned);
        end;
    else
        case fp_size of
            when 16 =>
                H(d) = FixedToFP{16, size}(S(d)[size-1:0], frac_bits, unsigned, fpcr,
                    FPRounding_TIEEVEN);
            when 32 =>
                S(d) = FixedToFP{32, size}(S(d)[size-1:0], frac_bits, unsigned, fpcr,
                    FPRounding_TIEEVEN);
            when 64 =>
                D(d) = FixedToFP{64, size}(D(d)[size-1:0], frac_bits, unsigned, fpcr,
                    FPRounding_TIEEVEN);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between floating-point and integer, Advanced SIMD)

Vector Convert between floating-point and integer converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	1	Vd				0	1	1	op	Q	M	0	Vm					
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let to_integer : boolean = (op[1] == '1');
let unsigned : boolean = (op[0] == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	1	Vd				0	1	1	op		Q	M	0	Vm			
opc1																																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && !InITBlock() then UnpredictableProcedure(); end;
let to_integer : boolean = (op[1] == '1');
let unsigned : boolean = (op[0] == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<dt1> Is the data type for the elements of the destination vector, encoded in "(size :: op)":

size	op	<dt1>
01	0x	F16
01	10	S16
01	11	U16
10	0x	F32
10	10	S32
10	11	U32

<dt2> Is the data type for the elements of the source vector, encoded in "(size :: op)":

size	op	<dt2>
01	00	S16
01	01	U16
01	1x	F16
10	00	S32
10	01	U32
10	1x	F32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    var result : bits(esize);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(m+r)[e*:esize];
            if to_integer then
                result = FPToFixed{esize, esize}(operand1, 0, unsigned, fpcr, FPRounding_ZERO);
            else
                result = FixedToFP{esize, esize}(operand1, 0, unsigned, fpcr, FPRounding_TIEEVEN);
            end;
            D(d+r)[e*:esize] = result;
        end;
    end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between half-precision and single-precision, Advanced SIMD)

Vector Convert between half-precision and single-precision converts each element in a vector from single-precision to half-precision floating-point, or from half-precision to single-precision, and places the results in a second vector.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0												
1	1	1	1	0	0	1	1	1	D	1	1	size			1			0			Vd			0			1		1		op		0		0		M		0		Vm		
opc1																Q																											

Encoding for the Half-precision to single-precision variant

Applies when (op == 1)

VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 0)

VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm>

Decode for all variants of this encoding

```
if size != '01' then Undefined(); end;
if op == '1' && Vd[0] == '1' then Undefined(); end;
if op == '0' && Vm[0] == '1' then Undefined(); end;
let half_to_single : boolean = (op == '1');
let esize : integer{} = 16;
let elements : integer = 4;
let m : integer = UInt(M::Vm);
let d : integer = UInt(D::Vd);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0				Vd		0	1	1	op	0	0	M	0			Vm	
opc1																Q															

Encoding for the Half-precision to single-precision variant

Applies when (op == 1)

VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 0)

VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm>

Decode for all variants of this encoding

```
if size != '01' then Undefined(); end;
if op == '1' && Vd[0] == '1' then Undefined(); end;
if op == '0' && Vm[0] == '1' then Undefined(); end;
let half_to_single : boolean = (op == '1');
let esize : integer{} = 16;
let elements : integer = 4;
let m : integer = UInt(M::Vm);
let d : integer = UInt(D::Vd);
```

Assembler Symbols

<c>	For the "A1 Half-precision to single-precision" and "A1 Single-precision to half-precision" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1 Half-precision to single-precision" and "T1 Single-precision to half-precision" variants: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for e = 0 to elements-1 do
        if half_to_single then
            Q(d>>1)[e*32] = FPConvert{32, 16}(Din(m)[e*16], fpcr);
        else
            D(d)[e*16] = FPConvert{16, 32}(Qin(m>>1)[e*32], fpcr);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (floating-point to integer, floating-point)

Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	1	1	0	x	Vd				1	0	size	1	1	M	0	Vm				
cond				o1								opc2				op															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (opc2 == 100 && size == 01)

```
VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (opc2 == 101 && size == 01)

```
VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (opc2 == 100 && size == 10)

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (opc2 == 101 && size == 10)

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (opc2 == 100 && size == 11)

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Encoding for the Double-precision scalar variant

Applies when (opc2 == 101 && size == 11)

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	1	D	1	1	1	1	0	x	Vd				1	0	size	1	1	M	0	Vm					
									o1								opc2				op											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (opc2 == 100 && size == 01)

VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (opc2 == 101 && size == 01)

VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (opc2 == 100 && size == 10)

VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (opc2 == 101 && size == 10)

VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (opc2 == 100 && size == 11)

VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Encoding for the Double-precision scalar variant

Applies when (opc2 == 101 && size == 11)

VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Floating-point data-processing* for the T32 instruction set, or *Floating-point data-processing* for the A32 instruction set.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 16 =>
                S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
        end;
    else
        case esize of
            when 16 =>
                H(d) = FixedToFP{16, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FixedToFP{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                D(d) = FixedToFP{64, 32}(S(m), 0, unsigned, fpcr, rounding);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (from single-precision to BFloat16, Advanced SIMD)

Vector Convert from single-precision to BFloat16 converts each 32-bit element in a vector from single-precision floating-point to BFloat16 format, and writes the result into a second vector. The result vector elements are half the width of the source vector elements.

Unlike the BFloat16 multiplication instructions, this instruction uses the Round to Nearest rounding mode, and can generate a floating-point exception that causes cumulative exception bits in the *FPSCR* to be set.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1
(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	0	1	1	0	Vd				0	1	opc2				1	M	0	Vm			
size												opc1				opc2				Q												

Encoding

VCVT{<c>}{<q>}.BFloat16.F32 <Dd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_AA32BF16)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	1	1	1	1	1	D	1	1	0	1	1	0	Vd				0	1	1				0	0	1	M	0	Vm			
size												opc1				opc2				Q														

Encoding

VCVT{<c>}{<q>}.BFloat16.F32 <Dd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>	For the "A1" variant: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1" variant: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
var operand : bits(128);
var result : bits(64);

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();

    operand = Q(m>>1);
    for e = 0 to 3 do
        let op : bits(32) = operand[e*32];
        result[e*16] = FPConvertBF(op, fpcr);
    end;
    D(d) = result;
end;
```

VCVT (integer to floating-point, floating-point)

Convert integer to floating-point converts a 32-bit integer to floating-point using the rounding mode specified by the *FPSCR*, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the *CPACR*, *NSACR*, *HCPTTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	1	0	0	0	Vd				1	0	size	op	1	M	0	Vm				
cond												o1																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

$\text{VCVT}\{\langle c \rangle\}\{\langle q \rangle\}.\text{F16}.\langle dt \rangle \langle Sd \rangle, \langle Sm \rangle$

Encoding for the Single-precision scalar variant

Applies when (size == 10)

$\text{VCVT}\{\langle c \rangle\}\{\langle q \rangle\}.\text{F32}.\langle dt \rangle \langle Sd \rangle, \langle Sm \rangle$

Encoding for the Double-precision scalar variant

Applies when (size == 11)

$\text{VCVT}\{\langle c \rangle\}\{\langle q \rangle\}.\text{F64}.\langle dt \rangle \langle Dd \rangle, \langle Sm \rangle$

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	1	0	0	0	Vd				1	0	size	op	1	M	0	Vm				
o1																opc2															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

$\text{VCVT}\{\langle c \rangle\}\{\langle q \rangle\}.\text{F16}.\langle dt \rangle \langle Sd \rangle, \langle Sm \rangle$

Encoding for the Single-precision scalar variant

Applies when (size == 10)

$\text{VCVT}\{\langle c \rangle\}\{\langle q \rangle\}.\text{F32}.\langle dt \rangle \langle Sd \rangle, \langle Sm \rangle$

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Floating-point data-processing](#) for the T32 instruction set, or [Floating-point data-processing](#) for the A32 instruction set.

Assembler Symbols

<c> See [Standard assembler syntax fields](#).
<q> See [Standard assembler syntax fields](#).
<dt> Is the data type for the operand, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 16 =>
                S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
        end;
    else
        case esize of
            when 16 =>
                H(d) = FixedToFP{16, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FixedToFP{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                D(d) = FixedToFP{64, 32}(S(m), 0, unsigned, fpcr, rounding);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTA (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest with Ties to Away converts each element in a vector from floating-point to integer using the Round to Nearest with Ties to Away rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd			0		0	0	0	op	Q	M	0	Vm			
												opc1																RM			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	1	Vd				0	0	0	0	op	Q	M	0	Vm			
opc1															RM																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock\(\)](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See [Standard assembler syntax fields](#).

<dt> Is the data type for the elements of the destination, encoded in “(op :: size)”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        D(d+r)[e*esize] = FPToFixed(esize, esize) (D(m+r)[e*esize],
                                                    0, unsigned, fpcr, rounding);
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTA (floating-point)

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	0	0	Vd			1		0	size	op	1	M	0	Vm				
										op1		RM																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	0	0	Vd				1		0	size	op	1	M	0	Vm			
										op1		RM																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```


CONSTRAINED UNPREDICTABLE behavior

If `InitTBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the destination, encoded in “op”:						
<table><tr><th>op</th><th><dt></th></tr><tr><td>0</td><td>U32</td></tr><tr><td>1</td><td>S32</td></tr></table>		op	<dt>	0	U32	1	S32
op	<dt>						
0	U32						
1	S32						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.						
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.						

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 16 =>
    S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
  when 32 =>
    S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
  when 64 =>
    S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTB

Convert to or from a half-precision value in the bottom half of a single-precision register does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1	sz	0	1	M	0	Vm			
cond										o1										T											

Encoding for the Half-precision to single-precision variant

Applies when (op == 0 && sz == 0)

VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Encoding for the Half-precision to double-precision variant

Applies when (op == 0 && sz == 1)

VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 1 && sz == 0)

VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Encoding for the Double-precision to half-precision variant

Applies when (op == 1 && sz == 1)

VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == '1' then 16 else 0);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1	sz	0	1	M	0	Vm			
o1															T																

Encoding for the Half-precision to single-precision variant

Applies when (op == 0 && sz == 0)

VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Encoding for the Half-precision to double-precision variant

Applies when (op == 0 && sz == 1)

VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 1 && sz == 0)

```
VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

Encoding for the Double-precision to half-precision variant

Applies when (op == 1 && sz == 1)

```
VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer = (if T == '1' then 16 else 0);
```

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    var hp : bits(16);
    let fpcr : FPCR_Type = EffectiveFPCR();
    if convert_from_half then
        hp = S(m)[lowbit+15:lowbit];
        if uses_double then
            D(d) = FPConvert{64, 16}(hp, fpcr);
        else
            S(d) = FPConvert{32, 16}(hp, fpcr);
        end;
    else
        if uses_double then
            hp = FPConvert{16, 64}(D(m), fpcr);
        else
            hp = FPConvert{16, 32}(S(m), fpcr);
        end;
        S(d)[lowbit+15:lowbit] = hp;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

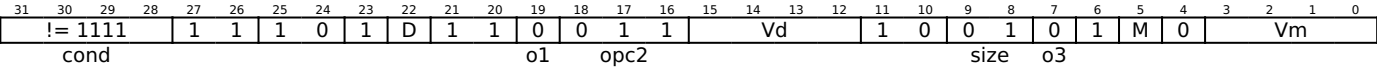
VCVTB (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result into the bottom half of a single precision register, preserving the top 16 bits of the destination register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the *FPSCR* that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the *FPSCR* to be set, or a synchronous exception to be taken, depending on the enable bits in the *FPSCR*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_AA32BF16)



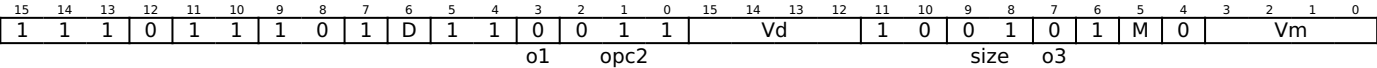
Encoding

```
VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

T1
(FEAT_AA32BF16)



Encoding

```
VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);

    S(d)[15:0] = FPConvertBF(S(m), EffectiveFPCR());
end;
```

VCVTM (Advanced SIMD)

Vector Convert floating-point to integer with Round towards -Infinity converts each element in a vector from floating-point to integer using the Round towards -Infinity rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd			0		0	1	1	op	Q	M	0	Vm			
opc1														RM																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

`VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
1	1	1	1	1	1	1	1	1	D	1	1	size			1			1			Vd			0		0		1		1		op		Q		M		0		Vm		
opc1														RM																												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

`VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>`

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock\(\)](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See [Standard assembler syntax fields](#).

<dt> Is the data type for the elements of the destination, encoded in “(op :: size)”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        D(d+r)[e*esize] = FPToFixed(esize, esize) (D(m+r)[e*esize],
                                                    0, unsigned, fpcr, rounding);
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTM (floating-point)

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	1	Vd			1		0	size	op	1	M	0	Vm				
										op1		RM																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	1	Vd			1		0	size	op	1	M	0	Vm				
										op1		RM																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InitTBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the destination, encoded in “op”:						
<table><tr><th>op</th><th><dt></th></tr><tr><td>0</td><td>U32</td></tr><tr><td>1</td><td>S32</td></tr></table>		op	<dt>	0	U32	1	S32
op	<dt>						
0	U32						
1	S32						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.						
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.						

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 16 =>
    S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
  when 32 =>
    S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
  when 64 =>
    S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTN (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest converts each element in a vector from floating-point to integer using the Round to Nearest rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd			0		0	0	1	op	Q	M	0	Vm			
opc1														RM																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	1		Vd			0	0	0	1	op	Q	M	0		Vm		
opc1														RM																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock\(\)](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See [Standard assembler syntax fields](#).

<dt> Is the data type for the elements of the destination, encoded in “(op :: size)”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        D(d+r)[e*esize] = FPToFixed(esize, esize) (D(m+r)[e*esize],
                                                    0, unsigned, fpcr, rounding);
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTN (floating-point)

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in a second register.

Depending on settings in the *CPACR*, *NSACR*, *HCPtr*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	0	1	Vd				1	0	size		op	1	M	0	Vm			
										op1		RM																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	0	1	Vd				1		0	size		op	1	M	0	Vm		
										op1		RM																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

- If `InitTBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the destination, encoded in “op”:						
<table><tr><th>op</th><th><dt></th></tr><tr><td>0</td><td>U32</td></tr><tr><td>1</td><td>S32</td></tr></table>		op	<dt>	0	U32	1	S32
op	<dt>						
0	U32						
1	S32						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.						
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.						

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 16 =>
    S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
  when 32 =>
    S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
  when 64 =>
    S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```

VCVTP (Advanced SIMD)

Vector Convert floating-point to integer with Round towards +Infinity converts each element in a vector from floating-point to integer using the Round towards +Infinity rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd			0		0	1	0	op	Q	M	0	Vm			
												opc1																RM			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	1		Vd			0	0	1	0	op	Q	M	0		Vm		
												opc1																RM			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock\(\)](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See [Standard assembler syntax fields](#).
 <dt> Is the data type for the elements of the destination, encoded in “(op :: size)”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
 <Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
 <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
 <Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  for e = 0 to elements-1 do
    D(d+r)[e*esize] = FPToFixed(esize, esize) (D(m+r)[e*esize],
                                                0, unsigned, fpcr, rounding);
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTP (floating-point)

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	0	Vd			1	0	size		op	1	M	0	Vm				
										op1		RM																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

`VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>`

Encoding for the Single-precision scalar variant

Applies when (size == 10)

`VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when (size == 11)

`VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>`

Decode for all variants of this encoding

```
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	0	Vd			1		0	size		op	1	M	0	Vm			
										op1		RM																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

`VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>`

Encoding for the Single-precision scalar variant

Applies when (size == 10)

`VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when (size == 11)

`VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>`

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

- If `InitTBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the destination, encoded in “op”: <table><tr><th>op</th><th><dt></th></tr><tr><td>0</td><td>U32</td></tr><tr><td>1</td><td>S32</td></tr></table>	op	<dt>	0	U32	1	S32
op	<dt>						
0	U32						
1	S32						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.						
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.						

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 16 =>
    S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
  when 32 =>
    S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
  when 64 =>
    S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```


VCVTR

Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by the *FPSCR* and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the *CPACR*, *NSACR*, *HCPTTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be *UNDEFINED*, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	1	1	0	x	Vd				1	0	size		0	1	M	0	Vm			
cond				o1								opc2				op															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (opc2 == 100 && size == 01)

```
VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (opc2 == 101 && size == 01)

```
VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (opc2 == 100 && size == 10)

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (opc2 == 101 && size == 10)

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (opc2 == 100 && size == 11)

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Encoding for the Double-precision scalar variant

Applies when (opc2 == 101 && size == 11)

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is *UNDEFINED*.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as *NOP*. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	1	D	1	1	1	1	0	x	Vd				1	0	size		0	1	M	0	Vm				
									o1				opc2				op															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (opc2 == 100 && size == 01)

VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (opc2 == 101 && size == 01)

VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (opc2 == 100 && size == 10)

VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (opc2 == 101 && size == 10)

VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (opc2 == 100 && size == 11)

VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Encoding for the Double-precision scalar variant

Applies when (opc2 == 101 && size == 11)

VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Floating-point data-processing* for the T32 instruction set, or *Floating-point data-processing* for the A32 instruction set.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 16 =>
                S(d) = FPToFixed{32, 16}(H(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
        end;
    else
        case esize of
            when 16 =>
                H(d) = FixedToFP{16, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 32 =>
                S(d) = FixedToFP{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                D(d) = FixedToFP{64, 32}(S(m), 0, unsigned, fpcr, rounding);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTT

Convert to or from a half-precision value in the top half of a single-precision register does one of the following:

- Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1	sz	1	1	M	0	Vm			
cond				o1								T																			

Encoding for the Half-precision to single-precision variant

Applies when (op == 0 && sz == 0)

```
VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

Encoding for the Half-precision to double-precision variant

Applies when (op == 0 && sz == 1)

```
VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

Encoding for the Single-precision to half-precision variant

Applies when (op == 1 && sz == 0)

```
VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

Encoding for the Double-precision to half-precision variant

Applies when (op == 1 && sz == 1)

```
VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == '1' then 16 else 0);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1	sz	1	1	M	0	Vm			
o1																T															

Encoding for the Half-precision to single-precision variant

Applies when (op == 0 && sz == 0)

```
VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

Encoding for the Half-precision to double-precision variant

Applies when (op == 0 && sz == 1)

```
VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

Encoding for the Single-precision to half-precision variant

Applies when (op == 1 && sz == 0)

```
VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

Encoding for the Double-precision to half-precision variant

Applies when (op == 1 && sz == 1)

```
VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer = (if T == '1' then 16 else 0);
```

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    var hp : bits(16);
    let fpcr : FPCR_Type = EffectiveFPCR();
    if convert_from_half then
        hp = S(m)[lowbit+15:lowbit];
        if uses_double then
            D(d) = FPConvert{64, 16}(hp, fpcr);
        else
            S(d) = FPConvert{32, 16}(hp, fpcr);
        end;
    else
        if uses_double then
            hp = FPConvert{16, 64}(D(m), fpcr);
        else
            hp = FPConvert{16, 32}(S(m), fpcr);
        end;
        S(d)[lowbit+15:lowbit] = hp;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

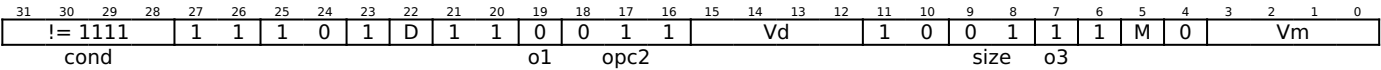
VCVTT (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result in the top half of a single-precision register, preserving the bottom 16 bits of the register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the *FPSCR* that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the *FPSCR* to be set, or a synchronous exception to be taken, depending on the enable bits in the *FPSCR*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_AA32BF16)



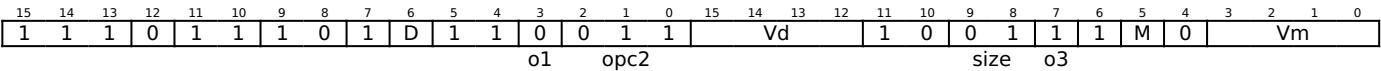
Encoding

```
VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

T1
(FEAT_AA32BF16)



Encoding

```
VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);

    S(d)[31:16] = FPConvertBF(S(m), EffectiveFPCR());
end;
```

VDIV

Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register. Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	0	0	Vn				Vd				1	0	size	N	0	M	0	Vm				
cond				o0				o1				o2																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VDIV{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VDIV{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VDIV{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	0	0	Vn				Vd				1	0	size	N	0	M	0	Vm				
				o0				o1				o2																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VDIV{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VDIV{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VDIV{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            H(d) = FPDIV{16}(H(n), H(m), fpcr);
        when 32 =>
            S(d) = FPDIV{32}(S(n), S(m), fpcr);
        when 64 =>
            D(d) = FPDIV{64}(D(n), D(m), fpcr);
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VDOT (by element)

BFloat16 floating-point indexed dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the indexed pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BFloat16 values in the first source vector. The instruction does not update the *FPSCR* exception status.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 1 1 1 1 1 1 0								0 D 0 0				Vn				Vd				1 1 0				1 N Q M 0				Vm			
op1								op2												op4				U							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32BF16)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	0	0	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
op1								op2				op4								U											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

operand2 = Din(m);
for r = 0 to regs-1 do
  operand1 = Din(n+r);
  result = Din(d+r);
  for e = 0 to 1 do
    let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
    let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
    let elt2_a : bits(16) = operand2[(2 * i + 0)*:16];
    let elt2_b : bits(16) = operand2[(2 * i + 1)*:16];
    let sum : bits(32) = FPAdd_BF16(BFMulH(elt1_a, elt2_a, fpcr),
                                      BFMulH(elt1_b, elt2_b, fpcr), fpcr);
    result[e*:32] = FPAdd_BF16(result[e*:32], sum, fpcr);
  end;
  D(d+r) = result;
end;
```

2026-03_rel 2026-03-26 20:48:11

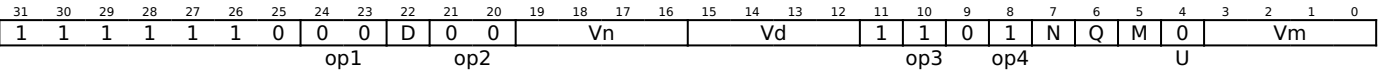
Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VDOT (vector)

BFloat16 floating-point (BF16) dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BF16 values in the first source vector. The instruction does not update the *FPSCR* exception status.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_AA32BF16)



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

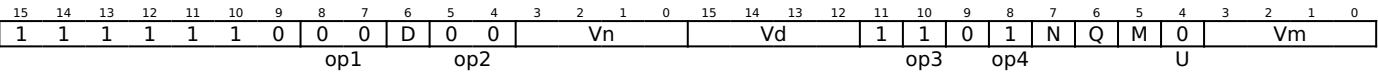
Applies when (Q == 1)

VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '1' then 2 else 1;
```

T1
(FEAT_AA32BF16)



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

for r = 0 to regs-1 do
  operand1 = Din(n+r);
  operand2 = Din(m+r);
  result = Din(d+r);
  for e = 0 to 1 do
    let elt1_a : bits(16) = operand1[(2 * e + 0)*:16];
    let elt1_b : bits(16) = operand1[(2 * e + 1)*:16];
    let elt2_a : bits(16) = operand2[(2 * e + 0)*:16];
    let elt2_b : bits(16) = operand2[(2 * e + 1)*:16];
    let sum : bits(32) = FPAdd_BF16(BFMulH(elt1_a, elt2_a, fpcr),
                                     BFMulH(elt1_b, elt2_b, fpcr), fpcr);
    result[e*:32] = FPAdd_BF16(result[e*:32], sum, fpcr);
  end;
  D(d+r) = result;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VDUP (general-purpose register)

Duplicate general-purpose register to vector duplicates an element from a general-purpose register into every element of the destination vector. The destination vector elements can be 8-bit, 16-bit, or 32-bit fields. The source element is the least significant 8, 16, or 32 bits of the general-purpose register. There is no distinction between data types.

Depending on settings in the *CPACR*, *NSACR*, and *HCPT* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	B	Q	0	Vd				Rt				1	0	1	1	D	0	E	1	(0)	(0)	(0)	(0)
cond											L																				

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VDUP{<c>}{<q>}.<size> <Dd>, <Rt>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VDUP{<c>}{<q>}.<size> <Qd>, <Rt>

Decode for all variants of this encoding

```
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
let regs : integer = if Q == '0' then 1 else 2;
if B::E == '11' then Undefined(); end;
let esize : integer{} = 32 >> UInt(B::E);
let elements : integer = 64 DIV esize;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	B	Q	0	Vd				Rt				1	0	1	1	D	0	E	1	(0)	(0)	(0)	(0)
L																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VDUP{<c>}{<q>}.<size> <Dd>, <Rt>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VDUP{<c>}{<q>}.<size> <Qd>, <Rt>

Decode for all variants of this encoding

```
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
let regs : integer = if Q == '0' then 1 else 2;
if B::E == '11' then Undefined(); end;
let esize : integer{} = 32 >> UInt(B::E);
let elements : integer = 64 DIV esize;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*. Arm strongly recommends that any VDUP instruction is unconditional, see *Conditional execution*.
- <q> See *Standard assembler syntax fields*.

<size>	The data size for the elements of the destination vector. It must be one of:
8	Encoded as [b, e] = 0b10.
16	Encoded as [b, e] = 0b01.
32	Encoded as [b, e] = 0b00.
<Dd>	The destination vector for a doubleword operation.
<Rt>	The Arm source register.
<Qd>	The destination vector for a quadword operation.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let scalar : bits(esize) = R(t)[esize-1:0];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*esize] = scalar;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VDUP (scalar)

Duplicate vector element to vector duplicates a single element of a vector into every element of the destination vector. The scalar, and the destination vector elements, can be any one of 8-bit, 16-bit, or 32-bit fields. There is no distinction between data types. For more information about scalars see *Advanced SIMD scalars*. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	imm4				Vd				1	1	0	0	0	Q	M	0	Vm			
opc																															

Encoding

Applies when (Q == 0)

VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x>

Encoding

Applies when (Q == 1)

VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x>

Decode for all variants of this encoding

```
if imm4 == 'x000' then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let lsb : integer{} = LowestSetBit(imm4[2:0]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(imm4[3:lsb+1]);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let elements : integer{} = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	imm4				Vd				1	1	0	0	0	Q	M	0	Vm			
opc																															

Encoding

Applies when (Q == 0)

VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x>

Encoding

Applies when (Q == 1)

VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x>

Decode for all variants of this encoding

```
if imm4 == 'x000' then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let lsb : integer{} = LowestSetBit(imm4[2:0]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(imm4[3:lsb+1]);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let elements : integer{} = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q> For the "T1" variant: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

<size>	The data size. It must be one of:
8	Encoded as imm4<0> = '1'. imm4<3:1> encodes the index[x] of the scalar.
16	Encoded as imm4<1:0> = '10'. imm4<3:2> encodes the index [x] of the scalar.
32	Encoded as imm4<2:0> = '100'. imm4<3> encodes the index [x] of the scalar.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm[x]>	The scalar. For details of how [x] is encoded, see the description of <size>.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let scalar : bits(esize) = D(m)[index*esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*esize] = scalar;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VEOR

Vector Bitwise Exclusive-OR performs a bitwise exclusive-OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			
U							size					opc							o1												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VEOR{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VEOR{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			
U									size				opc							o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VEOR{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VEOR{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled\(\);
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) XOR D(m+r);
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

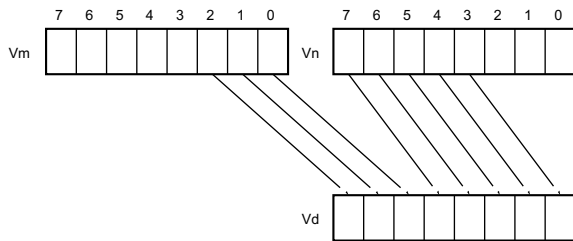
Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VEXT (byte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

The elements of the vectors are treated as being 8-bit fields. There is no distinction between data types.

The following figure shows an example of the operation of VEXT doubleword operation for $\text{imm} = 3$.



Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VEXT \(multibyte elements\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	1	1	Vn			Vd			imm4			N		Q	M	0	Vm					

Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

`VEXT{<c>}{<q>}.8 {<Dd>, }<Dn>, <Dm>, #<imm>`

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

`VEXT{<c>}{<q>}.8 {<Qd>, }<Qn>, <Qm>, #<imm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if Q == '0' && imm4[3] == '1' then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let position : integer{} = 8 * UInt(imm4);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

`VEXT{<c>}{<q>}.8 {<Dd>, }<Dn>, <Dm>, #<imm>`

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

`VEXT{<c>}{<q>}.8 {<Qd>, }<Qn>, <Qm>, #<imm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if Q == '0' && imm4[3] == '1' then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let position : integer{} = 8 * UInt(imm4);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<imm>	For the "A1 64-bit SIMD vector" and "T1 64-bit SIMD vector" variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 7, encoded in the "imm4" field. For the "A1 128-bit SIMD vector" and "T1 128-bit SIMD vector" variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 15, encoded in the "imm4" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    if quadword_operation then
        Q(d>>1) = (Q(m>>1)::Q(n>>1))[position+127:position];
    else
        D(d) = (D(m)::D(n))[position+63:position];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VEXT (multibyte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

This is a pseudo-instruction of [VEXT \(byte elements\)](#). This means:

- The encodings in this description are named to match the encodings of [VEXT \(byte elements\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VEXT \(byte elements\)](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VEXT{<c>}{<q>}.<size> {<Dd>, }<Dn>, <Dm>, #<imm>

is equivalent to

VEXT{<c>}{<q>}.8 {<Dd>, }<Dn>, <Dm>, #<imm*(size/8)>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VEXT{<c>}{<q>}.<size> {<Qd>, }<Qn>, <Qm>, #<imm>

is equivalent to

VEXT{<c>}{<q>}.8 {<Qd>, }<Qn>, <Qm>, #<imm*(size/8)>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VEXT{<c>}{<q>}.<size> {<Dd>, }<Dn>, <Dm>, #<imm>

is equivalent to

VEXT{<c>}{<q>}.8 {<Dd>, }<Dn>, <Dm>, #<imm*(size/8)>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VEXT{<c>}{<q>}.<size> {<Qd>, }<Qn>, <Qm>, #<imm>

is equivalent to

VEXT{<c>}{<q>}.8 {<Qd>, }<Qn>, <Qm>, #<imm*(size/8)>

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<size>	For the "A1 64-bit SIMD vector" and "T1 64-bit SIMD vector" variants: is the size of the operation, and can be one of 16 or 32. For the "A1 128-bit SIMD vector" and "T1 128-bit SIMD vector" variants: is the size of the operation, and can be one of 16, 32 or 64.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<imm>	For the "A1 64-bit SIMD vector" and "T1 64-bit SIMD vector" variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (128/<size>)-1. For the "A1 128-bit SIMD vector" and "T1 128-bit SIMD vector" variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (64/<size>)-1.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of [VEXT \(byte elements\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VEXT \(byte elements\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMA

Vector Fused Multiply Accumulate multiplies corresponding elements of two vectors, and accumulates the results into the elements of the destination vector. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the *CPACR*, *NSACR*, *HCPTTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	0	0	N	Q	M	1	Vm			
U							op					opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	0	Vn				Vd				1	0	size		N	0	M	0	Vm			
cond				o0					o1													op									

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	0	N	Q	M	1	Vm							
U								op												opc								o1							

Encoding for the 64-bit SIMD vector variant

Applies when (`Q == 0`)

VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (`Q == 1`)

VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	1	D	1	0	Vn			Vd			1	0	size		N	0	M	0	Vm						
o0								o1													op											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (`size == 01`)

VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (`size == 10`)

VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (`size == 11`)

VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional. For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see Standard assembler syntax fields. For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields.						
<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var oplelt : bits(esize) = D(n+r)[e*:esize];
                if opl_neg then oplelt = FPNeg{esize}(oplelt, fpcr); end;
                D(d+r)[e*:esize] = FPMulAdd{esize}(D(d+r)[e*:esize],
                                                    oplelt, D(m+r)[e*:esize], fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                let op16 : bits(16) = if opl_neg then FPNeg{16}(H(n), fpcr) else H(n);
                H(d) = FPMulAdd{16}(H(d), op16, H(m), fpcr);
            when 32 =>
                let op32 : bits(32) = if opl_neg then FPNeg{32}(S(n), fpcr) else S(n);
                S(d) = FPMulAdd{32}(S(d), op32, S(m), fpcr);
            when 64 =>
                let op64 : bits(64) = if opl_neg then FPNeg{64}(D(n), fpcr) else D(n);
                D(d) = FPMulAdd{64}(D(d), op64, D(m), fpcr);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

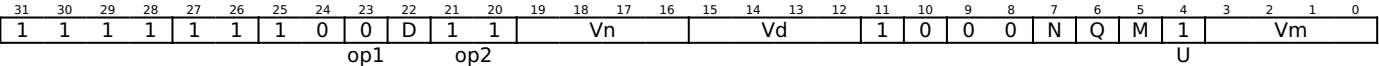
VFMAb, VFMAb (BFloat16, by scalar)

The BFloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and an indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the *FPSCR* to be set.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_AA32BF16)



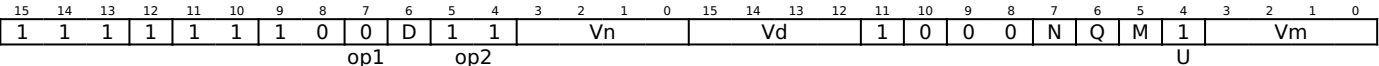
Encoding

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm[2:0]);
let i : integer = UInt(M::Vm[3]);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```

T1
(FEAT_AA32BF16)



Encoding

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm[2:0]);
let i : integer = UInt(M::Vm[3]);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```

Assembler Symbols

<bt>	Is the bottom or top element specifier, encoded in “Q”:						
<table><tr><th>Q</th><th><bt></th></tr><tr><td>0</td><td>B</td></tr><tr><td>1</td><td>T</td></tr></table>	Q	<bt>	0	B	1	T	
Q	<bt>						
0	B						
1	T						
<q>	See <i>Standard assembler syntax fields</i> .						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.						
<index>	Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.						

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

let operand1 : bits(128) = Q(n>>1);
let operand2 : bits(64) = D(m);
let operand3 : bits(128) = Q(d>>1);
var result : bits(128);

let element2 : bits(32) = operand2[i*16] :: Zeros{16};

for e = 0 to elements-1 do
  let element1 : bits(32) = operand1[(2 * e + sel)*16] :: Zeros{16};
  let addend : bits(32) = operand3[e*32];
  result[e*32] = FPMulAdd{32}(addend, element1, element2, fpcr);
end;

Q(d>>1) = result;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

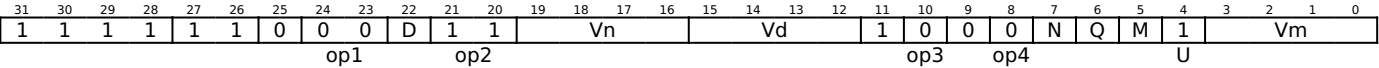
VFMAb, VFMAb (BFloat16, vector)

The Bfloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the *FPSCR* to be set.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_AA32BF16)



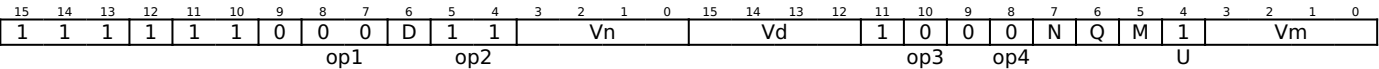
Encoding

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```

T1
(FEAT_AA32BF16)



Encoding

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 128 DIV 32;
let sel : integer = UInt(Q);
```

Assembler Symbols

<bt>	Is the bottom or top element specifier, encoded in “Q”:						
<table><tr><th>Q</th><th><bt></th></tr><tr><td>0</td><td>B</td></tr><tr><td>1</td><td>T</td></tr></table>	Q	<bt>	0	B	1	T	
Q	<bt>						
0	B						
1	T						
<q>	See <i>Standard assembler syntax fields</i> .						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

let operand1 : bits(128) = Q(n>>1);
let operand2 : bits(128) = Q(m>>1);
let operand3 : bits(128) = Q(d>>1);
var result : bits(128);

for e = 0 to elements-1 do
  let element1 : bits(32) = operand1[(2 * e + sel)*:16] :: Zeros{16};
  let element2 : bits(32) = operand2[(2 * e + sel)*:16] :: Zeros{16};
  let addend : bits(32) = operand3[e*:32];
  result[e*:32] = FPMulAdd{32}(addend, element1, element2, fpcr);
end;

Q(d>>1) = result;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMAL (by scalar)

Vector Floating-point Multiply-Add Long to accumulator (by scalar). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1 (FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	0	0	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
op1								S				U																			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(Vm[2:0]) else UInt(Vm[2:0]::M);

let index : integer = if Q == '1' then UInt(M::Vm[3]) else UInt(Vm[3]);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```

T1 (FEAT_FHM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	0	0	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
op1								S				U																			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(Vm[2:0]) else UInt(Vm[2:0]::M);

let index : integer = if Q == '1' then UInt(M::Vm[3]) else UInt(Vm[3]);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.
<index>	For the "A1 64-bit SIMD vector" and "T1 64-bit SIMD vector" variants: is the element index in the range 0 to 1, encoded in the "Vm<3>" field. For the "A1 128-bit SIMD vector" and "T1 128-bit SIMD vector" variants: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.

Operation

```
CheckAdvSIMDEnabled();
var operand1 : bits(datasize) ;
var operand2 : bits(datasize) ;
var operand3 : bits(64);
var result : bits(64);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);
let fpcr : FPCR_Type = StandardFPCR();

if Q=='0' then
    operand1 = S(n)[datasize-1:0];
    operand2 = S(m)[datasize-1:0];
else
    operand1 = D(n)[datasize-1:0];
    operand2 = D(m)[datasize-1:0];
end;
element2 = operand2[index*:(esize DIV 2)];
for r = 0 to regs-1 do
    operand3 = D(d+r);
    for e = 0 to 1 do
        element1 = operand1[(2*r+e)*:(esize DIV 2)];
        if sub_op then element1 = FPNeg{esize DIV 2}(element1, fpcr); end;
        result[e*esize] = FPMulAddH(operand3[e*esize], element1, element2, fpcr);
    end;
    D(d+r) = result;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMAL (vector)

Vector Floating-point Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

[ID_ISAR6](#).FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	0	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
S								op2			op3			op4			U														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(M::Vm) else UInt(Vm::M);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```

T1

(FEAT_FHM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	0	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
S								op2			op3			op4			U														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(M::Vm) else UInt(Vm::M);
let esize : integer{} = 32;
let regs : integer = 1 << UInt(Q);
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
```

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

var operand1 : bits(datasize);
var operand2 : bits(datasize);
var operand3 : bits(64);
var result : bits(64);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);

if Q=='0' then
    operand1 = S(n)[datasize-1:0];
    operand2 = S(m)[datasize-1:0];
else
    operand1 = D(n)[datasize-1:0];
    operand2 = D(m)[datasize-1:0];
end;

for r = 0 to regs-1 do
    operand3 = D(d+r);
    for e = 0 to 1 do
        element1 = operand1[(2*r+e)*:(esize DIV 2)];
        element2 = operand2[(2*r+e)*:(esize DIV 2)];
        if sub_op then element1 = FPNeg{esize DIV 2}(element1, fpcr); end;
        result[e*esize : e*esize+esize-1] = FPMulAddH(operand3[e*esize : e*esize+esize-1], element1, element2, fpcr);
    end;
    D(d+r) = result;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMS

Vector Fused Multiply Subtract negates the elements of one vector and multiplies them with the corresponding elements of another vector, adds the products to the corresponding elements of the destination vector, and places the results in the destination vector. The instruction does not round the result of the multiply before the addition.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn			Vd			1	1	0	0	N	Q	M	1	Vm					
U							op					opc										o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	0	Vn			Vd			1	0	size		N	1	M	0	Vm					
cond				o0					o1													op									

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	0	0	N	Q	M	1	Vm			
U								op												opc								o1			

Encoding for the 64-bit SIMD vector variant

Applies when (`Q == 0`)

VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (`Q == 1`)

VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	1	0	1	D	1	0	Vn				Vd				1	0	size		N	1	M	0	Vm							
o0								o1																op											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (`size == 01`)

VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (`size == 10`)

VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (`size == 11`)

VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional. For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see Standard assembler syntax fields. For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields.						
<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var oplelt : bits(esize) = D(n+r)[e*:esize];
                if opl_neg then oplelt = FPNeg{esize}(oplelt, fpcr); end;
                D(d+r)[e*:esize] = FPMulAdd{esize}(D(d+r)[e*:esize],
                                                    oplelt, D(m+r)[e*:esize], fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                let op16 : bits(16) = if opl_neg then FPNeg{16}(H(n), fpcr) else H(n);
                H(d) = FPMulAdd{16}(H(d), op16, H(m), fpcr);
            when 32 =>
                let op32 : bits(32) = if opl_neg then FPNeg{32}(S(n), fpcr) else S(n);
                S(d) = FPMulAdd{32}(S(d), op32, S(m), fpcr);
            when 64 =>
                let op64 : bits(64) = if opl_neg then FPNeg{64}(D(n), fpcr) else D(n);
                D(d) = FPMulAdd{64}(D(d), op64, D(m), fpcr);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMSL (by scalar)

Vector Floating-point Multiply-Subtract Long from accumulator (by scalar). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1 (FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	0	1	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
op1										S		U																			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(Vm[2:0]) else UInt(Vm[2:0]::M);

let index : integer = if Q == '1' then UInt(M::Vm[3]) else UInt(Vm[3]);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```

T1 (FEAT_FHM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	0	1	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
op1										S		U																			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(Vm[2:0]) else UInt(Vm[2:0]:M);

let index : integer = if Q == '1' then UInt(M::Vm[3]) else UInt(Vm[3]);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.
<index>	For the "A1 64-bit SIMD vector" and "T1 64-bit SIMD vector" variants: is the element index in the range 0 to 1, encoded in the "Vm<3>" field. For the "A1 128-bit SIMD vector" and "T1 128-bit SIMD vector" variants: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.

Operation

```
CheckAdvSIMDEnabled();
var operand1 : bits(datasize) ;
var operand2 : bits(datasize) ;
var operand3 : bits(64);
var result : bits(64);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);
let fpcr : FPCR_Type = StandardFPCR();

if Q=='0' then
    operand1 = S(n)[datasize-1:0];
    operand2 = S(m)[datasize-1:0];
else
    operand1 = D(n)[datasize-1:0];
    operand2 = D(m)[datasize-1:0];
end;
element2 = operand2[index*:(esize DIV 2)];
for r = 0 to regs-1 do
    operand3 = D(d+r);
    for e = 0 to 1 do
        element1 = operand1[(2*r+e)*:(esize DIV 2)];
        if sub_op then element1 = FPNeg{esize DIV 2}(element1, fpcr); end;
        result[e*esize] = FPMulAddH(operand3[e*esize], element1, element2, fpcr);
    end;
    D(d+r) = result;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMSL (vector)

Vector Floating-point Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

[ID_ISAR6](#).FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	1	D	1	0	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
S								op2				op3				op4				U											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(M::Vm) else UInt(Vm::M);
let esize : integer{} = 32;
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
let regs : integer = if Q == '0' then 1 else 2;
```

T1

(FEAT_FHM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	1	D	1	0	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
S								op2				op3				op4				U											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FHM) then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;

let d : integer = UInt(D::Vd);
let n : integer = if Q == '1' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if Q == '1' then UInt(M::Vm) else UInt(Vm::M);
let esize : integer{} = 32;
let regs : integer = 1 << UInt(Q);
let datasize : integer{} = 32 << UInt(Q);
let sub_op : boolean = S == '1';
```

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();

var operand1 : bits(datasize);
var operand2 : bits(datasize);
var operand3 : bits(64);
var result : bits(64);
var element1 : bits(esize DIV 2);
var element2 : bits(esize DIV 2);

if Q=='0' then
    operand1 = S(n)[datasize-1:0];
    operand2 = S(m)[datasize-1:0];
else
    operand1 = D(n)[datasize-1:0];
    operand2 = D(m)[datasize-1:0];
end;

for r = 0 to regs-1 do
    operand3 = D(d+r);
    for e = 0 to 1 do
        element1 = operand1[(2*r+e)*:(esize DIV 2)];
        element2 = operand2[(2*r+e)*:(esize DIV 2)];
        if sub_op then element1 = FPNeg{esize DIV 2}(element1, fpcr); end;
        result[e*esize : (e+1)*esize] = FPMulAddH(operand3[e*esize : (e+1)*esize], element1, element2, fpcr);
    end;
    D(d+r) = result;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFNMA

Vector Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	0	1	Vn				Vd				1	0	size		N	1	M	0	Vm			
cond				o0				o1												op											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	0	1	Vn				Vd				1	0	size		N	1	M	0	Vm			
o0								o1				op																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let op1_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let op16 : bits(16) = if op1_neg then FPNeg{16}(H(n), fpcr) else H(n);
            H(d) = FPMulAdd{16}(FPNeg{16}(H(d), fpcr), op16, H(m), fpcr);
        when 32 =>
            let op32 : bits(32) = if op1_neg then FPNeg{32}(S(n), fpcr) else S(n);
            S(d) = FPMulAdd{32}(FPNeg{32}(S(d), fpcr), op32, S(m), fpcr);
        when 64 =>
            let op64 : bits(64) = if op1_neg then FPNeg{64}(D(n), fpcr) else D(n);
            D(d) = FPMulAdd{64}(FPNeg{64}(D(d), fpcr), op64, D(m), fpcr);
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFNMMS

Vector Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				1	1	1	0	1	D	0	1	Vn				Vd				1 0				size	N	0	M	0	Vm			
cond				o0				o1												op												

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VFNMMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFNMMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFNMMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	0	1											1	0	size	N	0	M	0			Vm
				o0				o1				Vn				Vd				op				op				Vm			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VFNMMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFNMMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VFNMMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let op16 : bits(16) = if opl_neg then FPNeg{16}(H(n), fpcr) else H(n);
            H(d) = FPMulAdd{16}(FPNeg{16}(H(d), fpcr), op16, H(m), fpcr);
        when 32 =>
            let op32 : bits(32) = if opl_neg then FPNeg{32}(S(n), fpcr) else S(n);
            S(d) = FPMulAdd{32}(FPNeg{32}(S(d), fpcr), op32, S(m), fpcr);
        when 64 =>
            let op64 : bits(64) = if opl_neg then FPNeg{64}(D(n), fpcr) else D(n);
            D(d) = FPMulAdd{64}(FPNeg{64}(D(d), fpcr), op64, D(m), fpcr);
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VHADD

Vector Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. For rounded results, see [VRHADD](#)). The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	U	0	D	size			Vn			Vd			0 0		0 0		0	0	N	Q	M	0	Vm			
																					op						ol					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	U	1	1	1	1	0	D	size			Vn			Vd			0	0	0	0	N	Q	M	0	Vm					
															op						ol											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

<q>
<dt>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
See [Standard assembler syntax fields](#).

Is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if add then element1+element2 else element1-element2) >> 1;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VHSUB

Vector Halving Subtract subtracts the elements of the second operand from the corresponding elements of the first operand, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. There is no rounding version. The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size			Vn			Vd			0		0	1	0	N	Q	M	0	Vm			
																					op						ol				

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size			Vn			Vd			0	0	1	0	N	Q	M	0	Vm				
															op						ol										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

<q>
<dt>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
See [Standard assembler syntax fields](#).

Is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  for r = 0 to regs-1 do
    for e = 0 to elements-1 do
      let op1elt : bits(esize) = D(n+r)[e*:esize];
      let op2elt : bits(esize) = D(m+r)[e*:esize];
      let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
      let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
      let result : integer = (if add then element1+element2 else element1-element2) >> 1;
      D(d+r)[e*:esize] = result[esize-1:0];
    end;
  end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

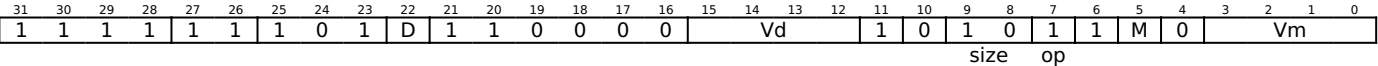
VINS

Vector move Insertion. This instruction copies the lower 16 bits of the 32-bit source SIMD&FP register into the upper 16 bits of the 32-bit destination SIMD&FP register, while preserving the values in the remaining bits.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_FP16)



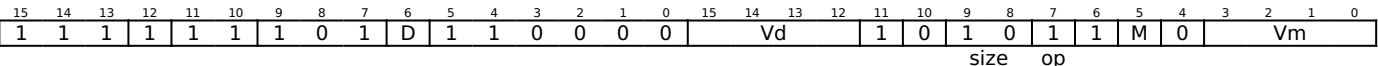
Encoding

VINS{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

T1
(FEAT_FP16)



Encoding

VINS{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

- If `InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    S(d)[31:16] = H(m);
end;
```

VJCVT

JavaScript Convert to signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo 2³², as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in *FPSCR*, the exception results in either a flag being set or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps*.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_JSCVT)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=	1	1	1	1	1	0	1	D	1	1	1	0	0	1		Vd				1	0	1	1	1	1	M	0		Vm		
cond								o1				opc2								size				o3							

Encoding

VJCVT{<q>}.S32.F64 <Sd>, <Dm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_JSCVT) then Undefined(); end;
if cond != '1110' then UnpredictableProcedure(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(M::Vm);
```

T1
(FEAT_JSCVT)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	1	0	0	1		Vd			1	0	1	1	1	1	M	0		Vm		
								o1				opc2								size				o3							

Encoding

VJCVT{<q>}.S32.F64 <Sd>, <Dm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_JSCVT) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fltval : bits(64) = D(m);
var intval : bits(32);
var Z : bit;
(intval, Z) = FPToFixedJS(fltval, EffectiveFPCR());
FPSCR()[31:28] = '0'::Z::'00';
S(d) = intval;
```

VLD1 (multiple single elements)

Load multiple single 1-element structures to one, two, three, or four registers loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) , [A3](#) and [A4](#)) and T32 ([T1](#) , [T2](#) , [T3](#) and [T4](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	1	0	Rn			Vd			0			1	1	1	size		align		Rm			
										L												itype									

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	1	0	Rn			Vd			1			0	1	0	size		align		Rm			
										L												itype									

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d+regs > 32**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	1	0	0	0	D	1	0	Rn			Vd			0			1	1	0	size		align		Rm				
L										itype																						

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d+regs > 32**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	1	0	0	0	D	1	0	Rn			Vd			0			0	1	0	size		align		Rm				
L										itype																						

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn			Vd			0	1	1	1	size		align		Rm					
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn			Vd			1	0	1	0	size		align		Rm					
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

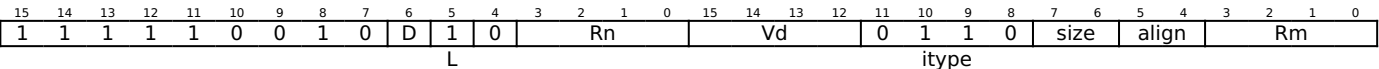
```
if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

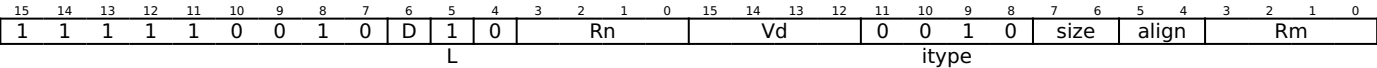
```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T4



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD1 \(multiple single elements\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c> For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", "A3 Post-indexed", "A4 Offset", and "A4 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", "T3 Post-indexed", "T4 Offset", and "T4 Post-indexed" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<size> Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32
11	64

<list> Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

- { <Dd> }
Single register. Selects the A1 and T1 encodings of the instruction.
- { <Dd>, <Dd+1> }
Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
- { <Dd>, <Dd+1>, <Dd+2> }
Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
- { <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }
Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64	64-bit alignment, encoded in the "align" field as 0b01.
128	128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.
256	256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var data : bits(ebytes*8);
            if ebytes != 8 then
                data = MemU{8*ebytes}(address);
            else
                if !IsAlignedSize(address, ebytes) && AlignmentEnforced() then
                    let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
                    AArch32_Abort(fault);
                end;

                if BigEndian(AccessType_ASIMD) then
                    data[31:0] = MemU{32}(address+4);
                    data[63:32] = MemU{32}(address);
                else
                    data[31:0] = MemU{32}(address);
                    data[63:32] = MemU{32}(address+4);
                end;
            end;

            D(d+r)[e*(8*ebytes)] = data;
            address = address + ebytes;
        end;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 8*regs;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD1 (single element to all lanes)

Load single 1-element structure and replicate to all lanes of one register loads one element from memory into every element of one or two vectors. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	0	0	size		T	a	Rm			
L											N																				

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' || (size == '00' && a == '1') then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let regs : integer = if T == '0' then 1 else 2;
let alignment : integer{} = if a == '0' then 1 else ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	0	0	size		T	a	Rm			
L										N																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' || (size == '00' && a == '1') then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let regs : integer = if T == '0' then 1 else 2;
let alignment : integer{} = if a == '0' then 1 else ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD1 (single element to all lanes)*.

Assembler Symbols

<c>

For the "A1 Offset" and "A1 Post-indexed" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

{ <Dd>[] }

Encoded in the "T" field as 0.

{ <Dd>[], <Dd+1>[] }

Encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

<align>

When <size> == 8, <align> must be omitted, otherwise it is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "a" field as 0. Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16

<align> is 16, meaning 16-bit alignment, encoded in the "a" field as 1.

<size> == 32

<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    let esize : integer{} = 8 * ebytes;
    let element : bits(esize) = MemU{8*ebytes}(address);
    let replicated_element : bits(64) = Replicate{}(element);
    for r = 0 to regs-1 do
        D(d+r) = replicated_element;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + ebytes;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD1 (single element to one lane)

Load single 1-element structure to one lane of one register loads one element from memory into one element of a register. Elements of the register that are not loaded are unchanged. For details of the addressing mode, see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) and [A3](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	0	0	0	index_align		Rm					
L											size										N										

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	1	0	0	index		align		Rm			
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

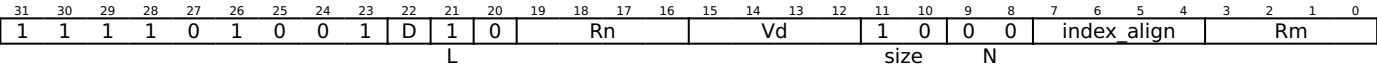
Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

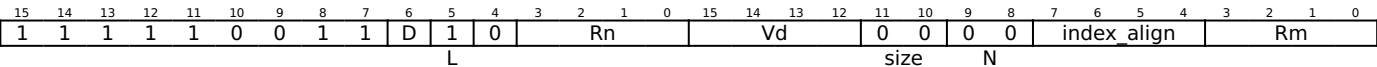
Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	1	0	0	index				align	Rm			
L											size																N					

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn			Vd			1	0	0	0	index_align				Rm					
L											size N																				

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)


```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", and "A3 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", and "T3 Post-indexed" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<size> Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list> Is a list containing the single 64-bit name of the SIMD&FP register holding the element. The list must be { <Dd>[<index>] }. The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8
 <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.
<size> == 16
 <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.
<size> == 32
 <index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> When <size> == 8, <align> must be omitted, otherwise it is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and the encoding depends on <size>:

<size> == 8
 Encoded in the "index_align<0>" field as 0.
<size> == 16
 Encoded in the "index_align<1:0>" field as 0b00.
<size> == 32
 Encoded in the "index_align<2:0>" field as 0b000.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16
 <align> is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as 0b01.
<size> == 32
 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as 0b011.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    D(d)[index*:(8*ebytes)] = MemU{8*ebytes}(address);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + ebytes;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

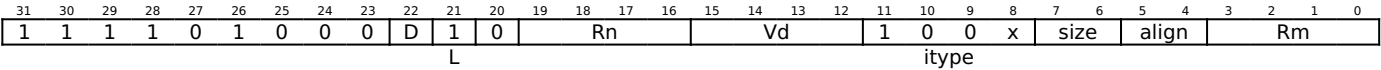
VLD2 (multiple 2-element structures)

Load multiple 2-element structures to two or four registers loads multiple 2-element structures from memory into two or four registers, with de-interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is loaded. For details of the addressing mode, see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

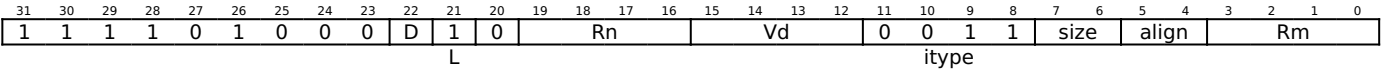
```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn			Vd			1	0	0	x	size		align		Rm					
L											itype																				

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

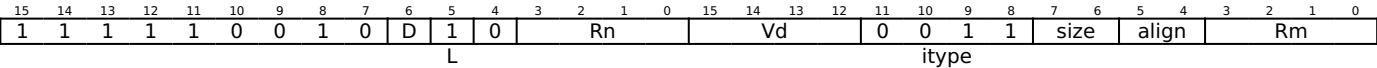
```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD2 \(multiple 2-element structures\)](#).
Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

- <c> For the "A1 Offset", "A1 Post-indexed", "A2 Offset", and "A2 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q> For the "T1 Offset", "T1 Post-indexed", "T2 Offset", and "T2 Post-indexed" variants: see [Standard assembler syntax fields](#).
- <size> See [Standard assembler syntax fields](#).
Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list>	Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of: { <Dd>, <Dd+1> } Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1000. { <Dd>, <Dd+2> } Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1001. { <Dd>, <Dd+1>, <Dd+2>, <Dd+3> } Three single-spaced registers. Selects the A2 and T2 encodings of the instruction. The register <Dd> is encoded in the "D:Vd" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<align>	Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are: 64 64-bit alignment, encoded in the "align" field as 0b01. 128 128-bit alignment, encoded in the "align" field as 0b10. 256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers. : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.
<Rm>	Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
        tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to pairs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e:(8*bytes)] = MemU{8*bytes}(address);
            D(d2+r)[e:(8*bytes)] = MemU{8*bytes}(address+bytes);
            address = address + 2*bytes;
        end;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 16*pairs;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD2 (single 2-element structure to all lanes)

Load single 2-element structure and replicate to all lanes of two registers loads one 2-element structure from memory into all lanes of two registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	0	1	size	T	a	Rm				
L										N																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let alignment : integer{} = if a == '0' then 1 else 2*ebytes;
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	0	1	size	T	a	Rm				
L										N																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let alignment : integer{} = if a == '0' then 1 else 2*ebytes;
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d2 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD2 \(single 2-element structure to all lanes\)](#).

Assembler Symbols

<c> For the "A1 Offset" and "A1 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<size> Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of two SIMD&FP registers. The list must be one of:

- { <Dd>[], <Dd+1>[] }
Single-spaced registers, encoded in the "T" field as 0.
- { <Dd>[], <Dd+2>[] }
Double-spaced registers, encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the "a" field as 0. Whenever <align> is present, the permitted values and encoding depend on <size>:

- <size> == 8
 <align> is 16, meaning 16-bit alignment, encoded in the "a" field as 1.
- <size> == 16
 <align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.
- <size> == 32
 <align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    let esize : integer{} = 8 * ebytes;
    let element1 : bits(esize) = MemU{8*ebytes}(address);
    let element2 : bits(esize) = MemU{8*ebytes}(address+ebytes);
    D(d) = Replicate{64}(element1);
    D(d2) = Replicate{64}(element2);

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 2*ebytes;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD2 (single 2-element structure to one lane)

Load single 2-element structure to one lane of two registers loads one 2-element structure from memory into corresponding elements of two registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see [Advanced SIMD addressing mode](#). Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) and [A3](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	0	0	1	index_align		Rm					
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	1	0	1	index_align				Rm						
L											size																N							

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

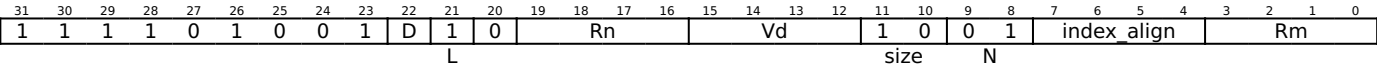
```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d2 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

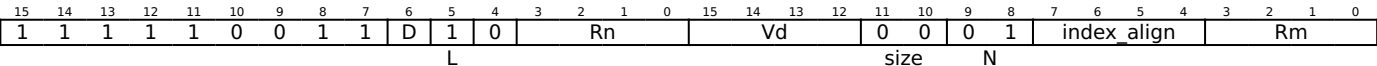
```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d2 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	1	0	1	index align				Rm			
L											size N																				

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

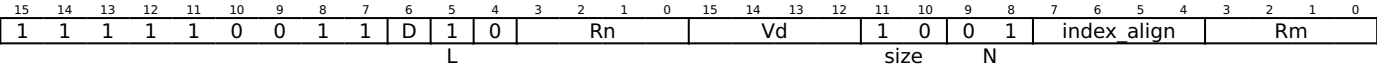
```
if size == '11' then See("VLD2 (single 2-element structure to one lane)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD2 \(single 2-element structure to one lane\)](#).

Assembler Symbols

- <c> For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", and "A3 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q> For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", and "T3 Post-indexed" variants: see [Standard assembler syntax fields](#).
- <size> See [Standard assembler syntax fields](#).
- Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list>	Is a list containing the 64-bit names of the two SIMD&FP registers holding the element. The list must be one of: <pre> { <Dd>[<index>], <Dd+1>[<index>] } Single-spaced registers, encoded as "spacing" = 0. { <Dd>[<index>], <Dd+2>[<index>] } Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8. </pre> The encoding of "spacing" depends on <size>: <pre> <size> == 16 "spacing" is encoded in the "index_align<1>" field. <size> == 32 "spacing" is encoded in the "index_align<2>" field. </pre> The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>: <pre> <size> == 8 <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field. <size> == 16 <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field. <size> == 32 <index> is 0 or 1, encoded in the "index_align<3>" field. </pre>
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<align>	Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on <size>: <pre> <size> == 8 Encoded in the "index_align<0>" field as 0. <size> == 16 Encoded in the "index_align<0>" field as 0. <size> == 32 Encoded in the "index_align<1:0>" field as 0b00. </pre> Whenever <align> is present, the permitted values and encoding depend on <size>: <pre> <size> == 8 <align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1. <size> == 16 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1. <size> == 32 <align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01. </pre> : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.
<Rm>	Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                         tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    D(d) [index*:(8*bytes)] = MemU{8*bytes}(address);
    D(d2) [index*:(8*bytes)] = MemU{8*bytes}(address+bytes);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 2*bytes;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD3 (multiple 3-element structures)

Load multiple 3-element structures to three registers loads multiple 3-element structures from memory into three registers, with de-interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is loaded. For details of the addressing mode, see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	1	0	Rn			Vd			0			1	0	x	size		align		Rm			
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if itype != '010x' then See("Related encodings"); end;
if size == '11' || align[1] == '1' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d3 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn			Vd			0			1	0	x	size		align		Rm			
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if itype != '010x' then See("Related encodings"); end;
if size == '11' || align[1] == '1' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD3 (multiple 3-element structures)*.
Related encodings: See *Advanced SIMD element or structure load/store* for the T32 instruction set, or *Advanced SIMD element or structure load/store* for the A32 instruction set.

Assembler Symbols

<c>	For the "A1 Offset" and "A1 Post-indexed" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1 Offset" and "T1 Post-indexed" variants: see <i>Standard assembler syntax fields</i> .								
<q>	See <i>Standard assembler syntax fields</i> .								
<size>	Is the data size, encoded in “size”: <table><tr><th>size</th><th><size></th></tr><tr><td>00</td><td>8</td></tr><tr><td>01</td><td>16</td></tr><tr><td>10</td><td>32</td></tr></table>	size	<size>	00	8	01	16	10	32
size	<size>								
00	8								
01	16								
10	32								
<list>	Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of: { <Dd>, <Dd+1>, <Dd+2> } Single-spaced registers, encoded in the "itype" field as 0b0100. { <Dd>, <Dd+2>, <Dd+4> } Double-spaced registers, encoded in the "itype" field as 0b0101. The register <Dd> is encoded in the "D:Vd" field.								
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.								
<align>	Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see <i>Unaligned data access</i> , and is encoded in the "align" field as 0b00. Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as 0b01. : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see <i>Advanced SIMD addressing mode</i> .								
<Rm>	Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field. For more information about <Rn>, !, and <Rm>, see <i>Advanced SIMD addressing mode</i> .								

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    for e = 0 to elements-1 do
        D(d)[e*:(8*ebytes)] = MemU{8*ebytes}(address);
        D(d2)[e*:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
        D(d3)[e*:(8*ebytes)] = MemU{8*ebytes}(address+2*ebytes);
        address = address + 3*ebytes;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 24;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD3 (single 3-element structure to all lanes)

Load single 3-element structure and replicate to all lanes of three registers loads one 3-element structure from memory into all lanes of three registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn			Vd			1	1	1	0	size	T	0	Rm						
L										N										a											

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```
if size == '11' || a == '1' then Undefined(); end;
let ebytes : integer() = 1 << UInt(size);
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d3 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn			Vd			1	1	1	0	size			T	0	Rm				
L										N										a											

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

$$\text{VLD3}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \langle \text{list} \rangle, [\langle \text{Rn} \rangle], \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if size == '11' || a == '1' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $d_3 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOF`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. If the instruction specifies writeback, the base register becomes `UNKNOWN`. This behavior does not affect any general-purpose registers.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD3 (single 3-element structure to all lanes)*.

Assembler Symbols

<c> For the "A1 Offset" and "A1 Post-indexed" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<code><size></code>	Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of three SIMD&FP registers. The list must be one of:

{ <Dd>[], <Dd+1>[], <Dd+2>[] }	Single-spaced registers, encoded in the "T" field as 0.
{ <Dd>[], <Dd+2>[], <Dd+4>[] }	Double-spaced registers, encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Alignment

Standard alignment rules apply, see [Alignment support](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let address : bits(32) = R(n);
    let esize : integer{} = ebytes * 8;
    let element1 : bits(esize) = MemU{8*ebytes}(address);
    let element2 : bits(esize) = MemU{8*ebytes}(address+ebytes);
    let element3 : bits(esize) = MemU{8*ebytes}(address+2*ebytes);

    D(d) = Replicate{64}(element1);
    D(d2) = Replicate{64}(element2);
    D(d3) = Replicate{64}(element3);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 3*ebytes;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD3 (single 3-element structure to one lane)

Load single 3-element structure to one lane of three registers loads one 3-element structure from memory into corresponding elements of three registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see [Advanced SIMD addressing mode](#). Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) and [A3](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	0	1	0	index_align		Rm					
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d3 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	1	1	0	index_align				Rm						
L											size																N							

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

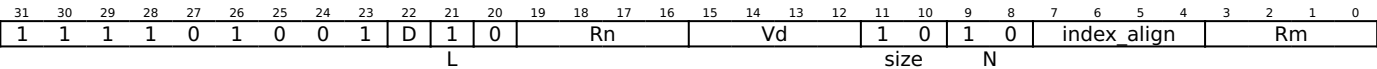
```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d3 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

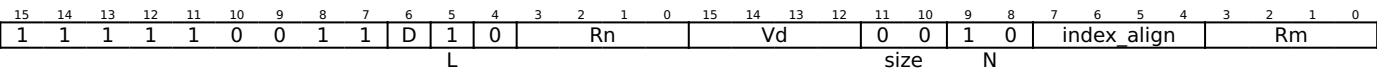
```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d3 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

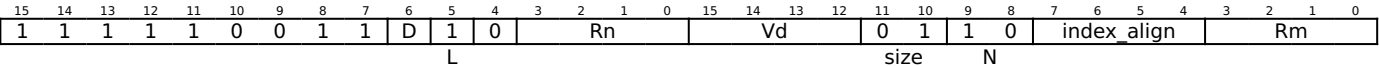
```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

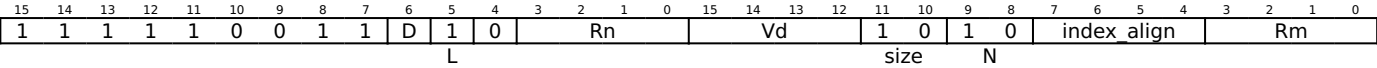
```
if size == '11' then See("VLD3 (single 3-element structure to one lane)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```


CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD3 \(single 3-element structure to one lane\)](#).

Assembler Symbols

<c>

<q>

<size>

For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", and "A3 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", and "T3 Post-indexed" variants: see [Standard assembler syntax fields](#).

See [Standard assembler syntax fields](#).

Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the three SIMD&FP registers holding the element. The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }
Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 8
"spacing" is encoded in the "index_align<0>" field.

<size> == 16
"spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.

<size> == 32
"spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Alignment

Standard alignment rules apply, see [Alignment support](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let address : bits(32) = R(n);
    D(d) [index*: (8*bytes)] = MemU{8*bytes}(address);
    D(d2) [index*: (8*bytes)] = MemU{8*bytes}(address+bytes);
    D(d3) [index*: (8*bytes)] = MemU{8*bytes}(address+2*bytes);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 3*bytes;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VLD4 (multiple 4-element structures)

Load multiple 4-element structures to four registers loads multiple 4-element structures from memory into four registers, with de-interleaving. For more information, see [Element and structure load/store instructions](#). Every element of each register is loaded. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	1	0	0	0	D	1	0	Rn			Vd			0			0	0	x	size	align	Rm						
L										itype																						

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if itype != '000x' then See("Related encodings"); end;
if size == '11' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn			Vd			0	0	0	x	size	align	Rm							
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if itype != '000x' then See("Related encodings"); end;
if size == '11' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d4 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD4 \(multiple 4-element structures\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c>	For the "A1 Offset" and "A1 Post-indexed" variants: see Standard assembler syntax fields . This encoding must be unconditional.								
	For the "T1 Offset" and "T1 Post-indexed" variants: see Standard assembler syntax fields .								
<q>	See Standard assembler syntax fields .								
<size>	Is the data size, encoded in "size": <table> <tr> <th>size</th><th><size></th></tr> <tr> <td>00</td><td>8</td></tr> <tr> <td>01</td><td>16</td></tr> <tr> <td>10</td><td>32</td></tr> </table>	size	<size>	00	8	01	16	10	32
size	<size>								
00	8								
01	16								
10	32								
<list>	Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of: { <Dd>, <Dd+1>, <Dd+2>, <Dd+3> } Single-spaced registers, encoded in the "itype" field as 0b0000. { <Dd>, <Dd+2>, <Dd+4>, <Dd+6> } Double-spaced registers, encoded in the "itype" field as 0b0001. The register <Dd> is encoded in the "D:Vd" field.								
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.								
<align>	Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access , and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are: 64 64-bit alignment, encoded in the "align" field as 0b01. 128 128-bit alignment, encoded in the "align" field as 0b10. 256 256-bit alignment, encoded in the "align" field as 0b11. : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.								
<Rm>	Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.								

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    for e = 0 to elements-1 do
        D(d)[e*:(8*ebytes)] = MemU{8*ebytes}(address);
        D(d2)[e*:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
        D(d3)[e*:(8*ebytes)] = MemU{8*ebytes}(address+2*ebytes);
        D(d4)[e*:(8*ebytes)] = MemU{8*ebytes}(address+3*ebytes);
        address = address + 4*ebytes;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 32;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD4 (single 4-element structure to all lanes)

Load single 4-element structure and replicate to all lanes of four registers loads one 4-element structure from memory into all lanes of four registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	1	1	size		T	a	Rm			
L										N																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' && a == '0' then Undefined(); end;
let ebytes : integer{} = if size == '11' then 4 else 1 << UInt(size);
let alignment : integer{} = (if a == '0' && size != '11' then 1
                             else 4 << (UInt(size[1]) + UInt(size[0])));
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	1	1	size		T	a	Rm			
L										N																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' && a == '0' then Undefined(); end;
let ebytes : integer{} = if size == '11' then 4 else 1 << UInt(size);
let alignment : integer{} = (if a == '0' && size != '11' then 1
                             else 4 << (UInt(size[1]) + UInt(size[0])));
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD4 (single 4-element structure to all lanes)*.

Assembler Symbols

<c> For the "A1 Offset" and "A1 Post-indexed" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in "size":

size	<size>
00	8
01	16
1x	32

<list> Is a list containing the 64-bit names of four SIMD&FP registers. The list must be one of:

- { <Dd>[], <Dd+1>[], <Dd+2>[], <Dd+3>[] }
Single-spaced registers, encoded in the "T" field as 0.
- { <Dd>[], <Dd+2>[], <Dd+4>[], <Dd+6>[] }
Double-spaced registers, encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "a" field as 0. Whenever <align> is present, the permitted values and encoding depend on <size>:

- <size> == 8
<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.
 - <size> == 16
<align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1.
 - <size> == 32
<align> can be 64 or 128. 64-bit alignment is encoded in the "a:size<0>" field as 0b10, and 128-bit alignment is encoded in the "a:size<0>" field as 0b11.
- : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    let esize : integer{} = ebytes * 8;
    let element1 : bits(esize) = MemU{8*ebytes}(address);
    let element2 : bits(esize) = MemU{8*ebytes}(address+ebytes);
    let element3 : bits(esize) = MemU{8*ebytes}(address+2*ebytes);
    let element4 : bits(esize) = MemU{8*ebytes}(address+3*ebytes);
    D(d) = Replicate{64}(element1);
    D(d2) = Replicate{64}(element2);
    D(d3) = Replicate{64}(element3);
    D(d4) = Replicate{64}(element4);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 4*ebytes;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD4 (single 4-element structure to one lane)

Load single 4-element structure to one lane of four registers loads one 4-element structure from memory into corresponding elements of four registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see [Advanced SIMD addressing mode](#). Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) and [A3](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	0	1	1	index_align		Rm					
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				0	1	1	1	index align		Rm					
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	0	1	1	index_align			Rm				
L											size											N									

Encoding for the Offset variant

Applies when (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				0	0	1	1	index_align				Rm							
L												size												N											

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

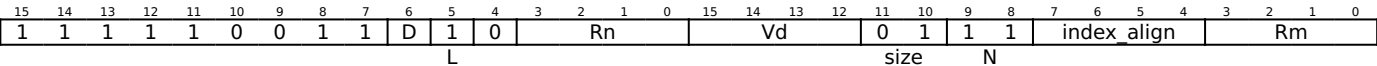
```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

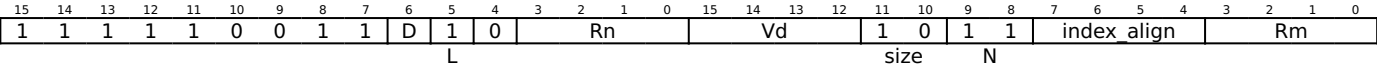
```
if size == '11' then See("VLD4 (single 4-element structure to one lane)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD4 (single 4-element structure to one lane)*.

Assembler Symbols

<c>

<q>

<size>

For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", and "A3 Post-indexed" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", and "T3 Post-indexed" variants: see *Standard assembler syntax fields*.

See *Standard assembler syntax fields*.

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>	Is a list containing the 64-bit names of the four SIMD&FP registers holding the element. The list must be one of: { <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] } Single-spaced registers, encoded as "spacing" = 0. { <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] } Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8. The encoding of "spacing" depends on <size>: <size> == 16 "spacing" is encoded in the "index_align<1>" field. <size> == 32 "spacing" is encoded in the "index_align<2>" field. The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>: <size> == 8 <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field. <size> == 16 <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field. <size> == 32 <index> is 0 or 1, encoded in the "index_align<3>" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<align>	Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on <size>: <size> == 8 Encoded in the "index_align<0>" field as 0. <size> == 16 Encoded in the "index_align<0>" field as 0. <size> == 32 Encoded in the "index_align<1:0>" field as 0b00. Whenever <align> is present, the permitted values and encoding depend on <size>: <size> == 8 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1. <size> == 16 <align> is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1. <size> == 32 <align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10. : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.
<Rm>	Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                         tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    D(d)[index*:(8*bytes)] = MemU{8*bytes}(address);
    D(d2)[index*:(8*bytes)] = MemU{8*bytes}(address+bytes);
    D(d3)[index*:(8*bytes)] = MemU{8*bytes}(address+2*bytes);
    D(d4)[index*:(8*bytes)] = MemU{8*bytes}(address+3*bytes);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 4*bytes;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLDM, VLDMDB, VLDMIA

Load Multiple SIMD&FP registers loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the alias [VPOP](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	1	x	x	x	x	x	x	x	0
cond				L												size				imm8											

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist> // (Preferred syntax)
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	0	imm8							
cond				L												size															

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist> // (Preferred syntax)

VLDmia{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd:D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	1	x	x	x	x	x	x	x	0
L												size						imm8													

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

VLDmdb{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist> // (Preferred syntax)

VLDmia{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

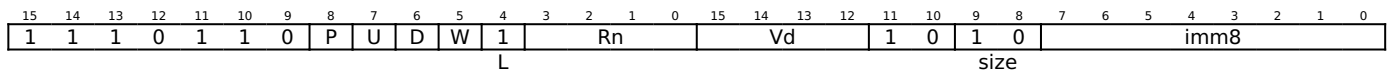
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist> // (Preferred syntax)

VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd::D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLDM](#).

Related encodings: See [Advanced SIMD and floating-point 64-bit move](#) for the T32 instruction set, or [Advanced SIMD and floating-point 64-bit move](#) for the A32 instruction set.

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<size>	An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.						
<dreglist>	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.						
!	Specifies base register writeback, encoded in "W": <table border="1"> <tr> <th>W</th><th>!</th></tr> <tr> <td>0</td><td>[absent]</td></tr> <tr> <td>1</td><td>[present]</td></tr> </table>	W	!	0	[absent]	1	[present]
W	!						
0	[absent]						
1	[present]						
<sreglist>	Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.						

Alias Conditions

Alias	Is preferred when
VPOP	P == '0' && U == '1' && W == '1' && Rn == '1101'

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckVFPEEnabled(TRUE);
  var address : bits(32) = if add then R(n) else R(n)-imm32;

  for r = 0 to regs-1 do
    if single_regs then
      S(d+r) = MemA{32}(address);
      address = address+4;
    else
      let word1 : bits(32) = MemA{32}(address);
      let word2 : bits(32) = MemA{32}(address+4);
      address = address+8;

      // Combine the word-aligned words in the correct order for current endianness.
      D(d+r) = if BigEndian(AccessType ASIMD) then word1::word2 else word2::word1;
    end;
  end;

  if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLDR (immediate)

Load SIMD&FP register (immediate) loads a single register from the Advanced SIMD and floating-point register file, using an address from a general-purpose register, with an optional offset.

Depending on settings in the *CPACR*, *NSACR*, *HCPTTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	1	U	D	0	1	!= 1111			Vd				1	0	size			imm8							
cond				P			W			L			Rn																		

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #<+/-><imm>}]
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VLDR{<c>}{<q>}.32 <Sd>, [<Rn> {, #<+/-><imm>}]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VLDR{<c>}{<q>}.64 <Dd>, [<Rn> {, #<+/-><imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	0	1	U	D	0	1	!= 1111				Vd				1	0	size			imm8							
P							W					L			Rn																	

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #<+/-><imm>}]
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VLDR{<c>}{<q>}.32 <Sd>, [<Rn> {, #<+/-><imm>}]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #<+/-><imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1"> <thead> <tr> <th>U</th><th>+/-</th></tr> </thead> <tbody> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </tbody> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Half-precision scalar" and "T1 Half-precision scalar" variants: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2. For the "A1 Double-precision scalar", "A1 Single-precision scalar", "T1 Double-precision scalar", and "T1 Single-precision scalar" variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.						
.32	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.						
.64	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let base : bits(32) = if n == 15 then AlignDownSize(PC32(),4) else R(n);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16 =>
            H(d) = MemA{16}(address);
        when 32 =>
            S(d) = MemA{32}(address);
        when 64 =>
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            // Combine the word-aligned words in the correct order for current endianness.
            D(d) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLDR (literal)

Load SIMD&FP register (literal) loads a single register from the Advanced SIMD and floating-point register file, using an address from the PC value and an immediate offset.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	1	U	D	0	1	1	1	1	1	Vd				1	0	size		imm8							
cond				P				W				L	Rn																		

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VLDR{<c>}{<q>}.16 <Sd>, <label> // (Preferred syntax)
VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VLDR{<c>}{<q>}.32 <Sd>, <label> // (Preferred syntax)
VLDR{<c>}{<q>}.32 <Sd>, [PC, #{+/-}<imm>]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VLDR{<c>}{<q>}.64 <Dd>, <label> // (Preferred syntax)
VLDR{<c>}{<q>}.64 <Dd>, [PC, #{+/-}<imm>]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	0	1	1	0	1	U	D	0	1	1	1	1	1	Vd				1	0	size		imm8												
							P				W				L				Rn																	

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VLDR{<c>}{<q>}.16 <Sd>, <label> // (Preferred syntax)
VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, <label> // (Preferred syntax)

VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, <label> // (Preferred syntax)

VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<label>	The label of the literal data item to be loaded. For the single-precision scalar or double-precision scalar variants: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020. For the half-precision scalar variant: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 2 in the range -510 to 510. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Half-precision scalar" and "T1 Half-precision scalar" variants: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2. For the "A1 Double-precision scalar", "A1 Single-precision scalar", "T1 Double-precision scalar", and "T1 Single-precision scalar" variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.						
.32	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.						
.64	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax .						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let base : bits(32) = if n == 15 then AlignDownSize(PC32(),4) else R(n);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16 =>
            H(d) = MemA{16}(address);
        when 32 =>
            S(d) = MemA{32}(address);
        when 64 =>
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            // Combine the word-aligned words in the correct order for current endianness.
            D(d) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMAX (floating-point)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			
U							op				opc																o1				

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm					
U										op				opc																o1			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && !InITBlock() then UnpredictableProcedure(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.						
	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
Floating-point maximum and minimum							
- max(+0.0, -0.0) = +0.0 - If any input is a NaN, the corresponding result element is the default NaN.							

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = D(n+r)[e*:esize];
            let op2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMMax{esize}(op1, op2, fpcr);
            else
                D(d+r)[e*:esize] = FPMMin{esize}(op1, op2, fpcr);
            end;
        end;
    end;
end;
```

VMAX (integer)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size										0	1	1	0	N	Q	M	0				
											opc										op							Vm			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	U	1	1	1	1	0	D	size	Vn					Vd					0	1	1	0	N	Q	M	0	Vm						
																opc								op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

<q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

<dt>

See *Standard assembler syntax fields*.

Is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if maximum then Max(element1, element2)
                                   else Min(element1, element2));
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VMAXNM

This instruction determines the floating-point maximum number.

It handles NaNs in consistence with the IEEE 754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point VMAX.

This instruction is not conditional.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			
U								op				opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMAXNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMAXNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn				Vd				1	0	size		N	0	M	0	Vm			
																								op							

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm				
U								op				opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMAXNM{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMAXNM{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

- If `InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn				Vd				1	0	size		N	0	M	0	Vm			
																op															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

- If `InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz":						
<table> <tr> <th>sz</th><th><dt></th></tr> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </table>		sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```

EncodingSpecificOperations();
CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
if advsimd then // Advanced SIMD instruction
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(n+r)[e*:esize];
            let operand2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMaxNum{esize}(operand1, operand2, fpcr);
            else
                D(d+r)[e*:esize] = FPMinNum{esize}(operand1, operand2, fpcr);
            end;
        end;
    end;
else // VFP instruction
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            if maximum then
                H(d) = FPMaxNum{16}(H(n), H(m), fpcr);
            else
                H(d) = FPMinNum{16}(H(n), H(m), fpcr);
            end;
        when 32 =>
            if maximum then
                S(d) = FPMaxNum{32}(S(n), S(m), fpcr);
            else
                S(d) = FPMinNum{32}(S(n), S(m), fpcr);
            end;
        when 64 =>
            if maximum then
                D(d) = FPMaxNum{64}(D(n), D(m), fpcr);
            else
                D(d) = FPMinNum{64}(D(n), D(m), fpcr);
            end;
    end;
end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMIN (floating-point)

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			
U							op				opc																o1				

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm					
U										op				opc																o1			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- <dt>

Is the data type for the elements of the vectors, encoded in "sz":

sz	<dt>
0	F32
1	F16
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- Floating-point minimum

- min(+0.0, -0.0) = -0.0
 - If any input is a NaN, the corresponding result element is the default NaN.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = D(n+r)[e*:esize];
            let op2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMMax{esize}(op1, op2, fpcr);
            else
                D(d+r)[e*:esize] = FPMMin{esize}(op1, op2, fpcr);
            end;
        end;
    end;
end;
```


VMIN (integer)

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn				Vd				0	1	1	0	N	Q	M	1	Vm				
											opc										op										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	0	D	size	Vn					Vd					0	1	1	0	N	Q	M	1	Vm				
											opc										op												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if maximum then Max(element1, element2)
                                   else Min(element1, element2));
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VMINNM

This instruction determines the floating point minimum number.

It handles NaNs in consistence with the IEEE 754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point `VMIN`.

This instruction is not conditional.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			
U								op				opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMINNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMINNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn				Vd				1	0	size		N	1	M	0	Vm			
																								op							

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm				
U								op				opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMINNM{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMINNM{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

- If `InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn				Vd				1	0	size		N	1	M	0	Vm			
op																															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

- If `InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz":						
<table> <tr> <th>sz</th><th><dt></th></tr> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </table>		sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```

EncodingSpecificOperations();
CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
if advsimd then // Advanced SIMD instruction
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(n+r)[e*:esize];
            let operand2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMaxNum{esize}(operand1, operand2, fpcr);
            else
                D(d+r)[e*:esize] = FPMinNum{esize}(operand1, operand2, fpcr);
            end;
        end;
    end;
else // VFP instruction
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            if maximum then
                H(d) = FPMaxNum{16}(H(n), H(m), fpcr);
            else
                H(d) = FPMinNum{16}(H(n), H(m), fpcr);
            end;
        when 32 =>
            if maximum then
                S(d) = FPMaxNum{32}(S(n), S(m), fpcr);
            else
                S(d) = FPMinNum{32}(S(n), S(m), fpcr);
            end;
        when 64 =>
            if maximum then
                D(d) = FPMaxNum{64}(D(n), D(m), fpcr);
            else
                D(d) = FPMinNum{64}(D(n), D(m), fpcr);
            end;
    end;
end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLA (by scalar)

Vector Multiply Accumulate multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector. For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	!= 11					Vn					Vd		0	0	0	F	N	1	M	0		Vm
size												op																			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
  Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	!= 11					Vn					Vd		0	0	0	F	N	1	M	0		Vm
size												op																			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
  Undefined();
end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in “(F :: size)”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                    fpcr),
                                                                    fpcr));
                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLA (floating-point)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1				1	0	1	N	Q	M	1	Vm			
U							op				opc										o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	0	0	Vn				Vd				1	0	size		N	0	M	0	Vm			
cond				o0				o1														op									

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			
U								op												opc								o1			

Encoding for the 64-bit SIMD vector variant

Applies when (`Q == 0`)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (`Q == 1`)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	1	0	0	D	0	0	Vn				Vd				1	0	size		N	0	M	0	Vm							
o0								o1																op											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (`size == 01`)

VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (`size == 10`)

VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (`size == 11`)

VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional. For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see Standard assembler syntax fields. For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields.						
<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in “sz”:						
<table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>		sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var addend : bits(esize) = FPMul{}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
                if !add then addend = FPNeg{esize}(addend, fpcr); end;
                D(d+r)[e*:esize] = FPAdd{esize}(D(d+r)[e*:esize], addend, fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                var addend16 : bits(16) = FPMul{}(H(n), H(m), fpcr);
                if !add then addend16 = FPNeg{16}(addend16, fpcr); end;
                H(d) = FPAdd{16}(H(d), addend16, fpcr);
            when 32 =>
                var addend32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
                if !add then addend32 = FPNeg{32}(addend32, fpcr); end;
                S(d) = FPAdd{32}(S(d), addend32, fpcr);
            when 64 =>
                var addend64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
                if !add then addend64 = FPNeg{64}(addend64, fpcr); end;
                D(d) = FPAdd{64}(D(d), addend64, fpcr);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLA (integer)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and adds the products to the corresponding elements of the destination vector.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	0	0	D	size			Vn				Vd				1		0		0		1	N	Q	M	0	Vm			
op										opc										o1															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	1	1	1	0	D	size				Vn				Vd				1	0	0	1	N	Q	M	0	Vm			
op										opc										o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<dt>	Is the data type for the elements of the operands, encoded in “size”:
size	<dt>
00	I8
01	I16
10	I32

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VMLAL (by scalar)

Vector Multiply Accumulate Long multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	U	1	D	!= 11		Vn				Vd				0		0	1	0	N	1	M	0	Vm			
size										op																						

Encoding

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	1	D	!= 11				Vn				Vd				0	0	1	0	N	1	M	0	Vm			
size										op																							

Encoding

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c> For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <dt> Is the data type for the scalar and the elements of the operand vector, encoded in “(U :: size)”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd>	Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm[x]>	Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                                   fpcr),
                                                                                   fpcr));
                D(d+r)[e*esize] = FPAdd{esize}(Din(d+r)[e*esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*(2*esize)] = Qin(d>>1)[e*(2*esize)] + addend;
                else
                    D(d+r)[e*esize] = Din(d+r)[e*esize] + addend;
                end;
            end;
        end;
    end;
end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

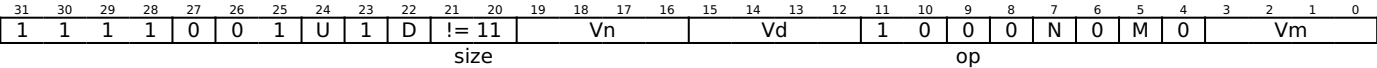
VMLAL (integer)

Vector Multiply Accumulate Long multiplies corresponding elements in two vectors, and add the products to the corresponding element of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



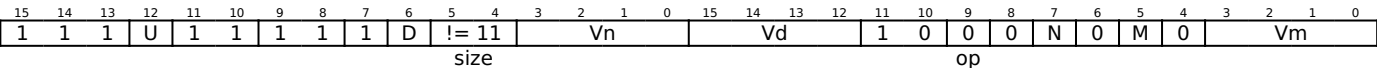
Encoding

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

T1



Encoding

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c>For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.
- <q>See *Standard assembler syntax fields*.
- <dt>Is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32
- <Qd>Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLS (by scalar)

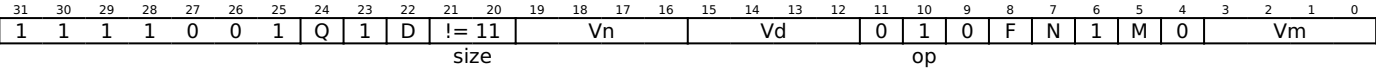
Vector Multiply Subtract multiplies elements of a vector by a scalar, and either subtracts the products from corresponding elements of the destination vector.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x>

Encoding for the 128-bit SIMD vector variant

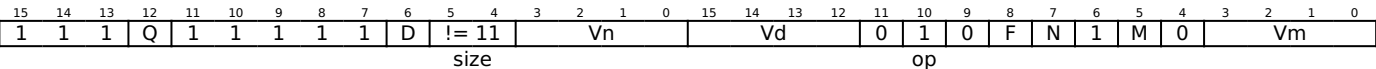
Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
  Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
  Undefined();
end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in “(F :: size)”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                    fpcr),
                                                                    fpcr));
                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLS (floating-point)

Vector Multiply Subtract multiplies corresponding elements in two vectors, subtracts the products from corresponding elements of the destination vector, and places the results in the destination vector.

Note

Arm recommends that software does not use the `VMLS` instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the `CPACR`, `NSACR`, `HCPTR`, and `FPEXC` registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			
U							op				opc								o1												

Encoding for the 64-bit SIMD vector variant

Applies when (`Q == 0`)

`VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (`Q == 1`)

`VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	0	0	Vn				Vd				1	0	size		N	1	M	0	Vm			
cond				o0				o1				op																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (`size == 01`)

`VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>`

Encoding for the Single-precision scalar variant

Applies when (`size == 10`)

`VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when (`size == 11`)

VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			
U									op			opc										ol									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	0	0	Vn				Vd				1	0	size	N	1	M	0	Vm				
o0									o1			op																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see <i>Standard assembler syntax fields</i> . For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the vectors, encoded in “sz”: <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var addend : bits(esize) = FPMul{}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
                if !add then addend = FPNeg{esize}(addend, fpcr); end;
                D(d+r)[e*:esize] = FPAdd{esize}(D(d+r)[e*:esize], addend, fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                var addend16 : bits(16) = FPMul{}(H(n), H(m), fpcr);
                if !add then addend16 = FPNeg{16}(addend16, fpcr); end;
                H(d) = FPAdd{16}(H(d), addend16, fpcr);
            when 32 =>
                var addend32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
                if !add then addend32 = FPNeg{32}(addend32, fpcr); end;
                S(d) = FPAdd{32}(S(d), addend32, fpcr);
            when 64 =>
                var addend64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
                if !add then addend64 = FPNeg{64}(addend64, fpcr); end;
                D(d) = FPAdd{64}(D(d), addend64, fpcr);
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLS (integer)

Vector Multiply Subtract multiplies corresponding elements in two vectors, and subtracts the products from the corresponding elements of the destination vector.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	0	D	size				Vn				Vd				1	0	0	1	N	Q	M	0	Vm			
op										opc										o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	1	1	0	D	size				Vn				Vd				1	0	0	1	N	Q	M	0	Vm			
op										opc										o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VMLSL (by scalar)

Vector Multiply Subtract Long multiplies elements of a vector by a scalar, and subtracts the products from corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	U	1	D	!= 11				Vn				Vd				0	1	1	0	N	1	M	0	Vm			
size										op																							

Encoding

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	1	D	!= 11				Vn				Vd				0	1	1	0	N	1	M	0	Vm			
size										op																							

Encoding

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c> For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <dt> Is the data type for the scalar and the elements of the operand vector, encoded in “(U :: size)”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd>	Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm[x]>	Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                                   fpcr),
                                                                                   fpcr));
                D(d+r)[e*esize] = FPAdd{esize}(Din(d+r)[e*esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*(2*esize)] = Qin(d>>1)[e*(2*esize)] + addend;
                else
                    D(d+r)[e*esize] = Din(d+r)[e*esize] + addend;
                end;
            end;
        end;
    end;
end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

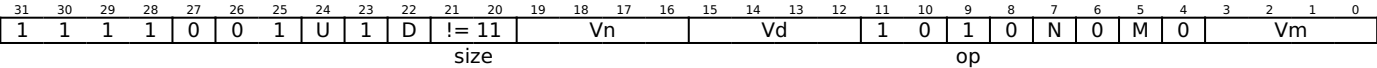
VMLSL (integer)

Vector Multiply Subtract Long multiplies corresponding elements in two vectors, and subtract the products from the corresponding elements of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



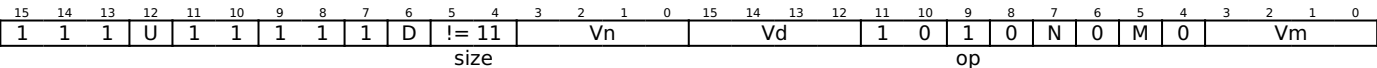
Encoding

```
VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

T1



Encoding

```
VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>

For the "T1" variant: see *Standard assembler syntax fields*.
- <dt>

See *Standard assembler syntax fields*.
- Is the data type for the elements of the operands, encoded in “(U :: size)”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VMMLA

BFloat16 floating-point matrix multiply-accumulate. This instruction multiplies the 2x4 matrix of BF16 values in the first 128-bit source vector by the 4x2 BF16 matrix in the second 128-bit source vector. The resulting 2x2 single-precision matrix product is then added destructively to the 2x2 single-precision matrix in the 128-bit destination vector. This is equivalent to performing a 4-way dot product per destination element. The instruction does not update the *FPSCR* exception status.

Note

Arm expects that the VMMLA instruction will deliver a peak BF16 multiply throughput that is at least as high as can be achieved using two VDOT instructions, with a goal that it should have significantly higher throughput.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	0	0	Vn				Vd				1	1	0	0	N	1	M	0	Vm			
op1						op2						op3				op4				Q		U									

Encoding

VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 2;
```

T1

(FEAT_AA32BF16)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	0	0	0	D	0	0	Vn				Vd				1	1	0	0	N	1	M	0	Vm					
op1						op2										op3				op4				Q		U							

Encoding

VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32BF16) then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 2;
```

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();

let operand1 : bits(128) = Q(n>>1);
let operand2 : bits(128) = Q(m>>1);
let acc : bits(128) = Q(d>>1);
let fpcr : FPCR_Type = EffectiveFPCR();

Q(d>>1) = BFMatMulAddH(acc, operand1, operand2, fpcr);
```


VMOV (between general-purpose register and half-precision)

Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in the bottom 16 bits of a 32-bit SIMD&FP register to the bottom 16 bits of a general-purpose register, or the value held in the bottom 16 bits of a general-purpose register to the bottom 16 bits of a 32-bit SIMD&FP register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1 1 1 0 0 0 0				op	Vn							Rt				1 0 0 1				N	(0) (0)		1	(0) (0)		(0)	(0)
cond																															

Encoding for the From general-purpose register variant

Applies when (op == 0)

VMOV{<c>}{<q>}.F16 <Sn>, <Rt>

Encoding for the To general-purpose register variant

Applies when (op == 1)

VMOV{<c>}{<q>}.F16 <Rt>, <Sn>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if cond != '1110' then UnpredictableProcedure(); end;
let to_arm_register : boolean = (op == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(Vn::N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

(FEAT_FP16)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	0	op	Vn				Rt				1	0	0	1	N	(0)	(0)	1	(0)	(0)	(0)	(0)

Encoding for the From general-purpose register variant

Applies when (op == 0)

VMOV{<c>}{<q>}.F16 <Sn>, <Rt>

Encoding for the To general-purpose register variant

Applies when (op == 1)

VMOV{<c>}{<q>}.F16 <Rt>, <Sn>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
let to_arm_register : boolean = (op == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(Vn::N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sn>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
<Rt>	Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    if to_arm_register then
        R(t) = Zeros{16} :: S(n)[15:0];
    else
        S(n) = Zeros{16} :: R(t)[15:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (between general-purpose register and single-precision)

Copy a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in a 32-bit SIMD&FP register to a general-purpose register, or the value held in a general-purpose register to a 32-bit SIMD&FP register.

Depending on settings in the *CPACR*, *NSACR*, *HCPtr*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	0	0	op	Vn				Rt				1	0	1	0	N	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																															

Encoding for the From general-purpose register variant

Applies when (op == 0)

VMOV{<c>}{<q>}<Sn>, <Rt>

Encoding for the To general-purpose register variant

Applies when (op == 1)

VMOV{<c>}{<q>}<Rt>, <Sn>

Decode for all variants of this encoding

```
let to_arm_register : boolean = (op == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(Vn::N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	0	op	Vn				Rt				1	0	1	0	N	(0)	(0)	1	(0)	(0)	(0)	(0)

Encoding for the From general-purpose register variant

Applies when (op == 0)

VMOV{<c>}{<q>}<Sn>, <Rt>

Encoding for the To general-purpose register variant

Applies when (op == 1)

VMOV{<c>}{<q>}<Rt>, <Sn>

Decode for all variants of this encoding

```
let to_arm_register : boolean = (op == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(Vn::N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Sn>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
<Rt>	Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    if to_arm_register then
         $\underline{R}(t) = \underline{S}(n)$ ;
    else
         $\underline{S}(n) = \underline{R}(t)$ ;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (between two general-purpose registers and a doubleword floating-point register)

Copy two general-purpose registers to or from a SIMD&FP register copies two words from two general-purpose registers into a doubleword register in the Advanced SIMD and floating-point register file, or from a doubleword register in the Advanced SIMD and floating-point register file to two general-purpose registers.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	0	0	1	0	op	Rt2				Rt				1	0	1	1	0	0	M	1	Vm			
cond				D																size		opc2		o3							

Encoding for the From general-purpose registers variant

Applies when (op == 0)

VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>

Encoding for the To general-purpose registers variant

Applies when (op == 1)

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>

Decode for all variants of this encoding

```
let to_arm_registers : boolean = (op == '1');
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(M::Vm);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	0	0	0	1	0	op	Rt2				Rt				1	0	1	1	0	0	M	1	Vm							
										D																size		opc2		o3					

Encoding for the From general-purpose registers variant

Applies when (op == 0)

VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>

Encoding for the To general-purpose registers variant

Applies when (op == 1)

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>

Decode for all variants of this encoding

```
let to_arm_registers : boolean = (op == '1');
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(M::Vm);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VMOV \(between two general-purpose registers and a doubleword floating-point register\)](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Dm>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "M:Vm" field.
<Rt>	Is the first general-purpose register that <Dm>[31:0] will be transferred to or from, encoded in the "Rt" field.
<Rt2>	Is the second general-purpose register that <Dm>[63:32] will be transferred to or from, encoded in the "Rt2" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    if to_arm_registers then
        R(t) = D(m)[31:0];
        R(t2) = D(m)[63:32];
    else
        D(m)[31:0] = R(t);
        D(m)[63:32] = R(t2);
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (between two general-purpose registers and two single-precision registers)

Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers transfers the contents of two consecutively numbered single-precision Floating-point registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision Floating-point registers. The general-purpose registers do not have to be contiguous.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	0	0	1	0	op	Rt2				Rt				1	0	1	0	0	0	M	1	Vm			
cond								D												size				opc2				o3			

Encoding for the From general-purpose registers variant

Applies when (op == 0)

VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>

Encoding for the To general-purpose registers variant

Applies when (op == 1)

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>

Decode for all variants of this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(Vm::M);
let m2 : integer = m + 1;
let to_arm_registers : boolean = (op == '1');
if t == 15 || t2 == 15 || m == 31 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	1	0	op	Rt2				Rt				1	0	1	0	0	0	M	1	Vm			
								D												size				opc2				o3			

Encoding for the From general-purpose registers variant

Applies when (op == 0)

VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>

Encoding for the To general-purpose registers variant

Applies when (op == 1)

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>

Decode for all variants of this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(Vm:M);
let m2 : integer = m + 1;
let to_arm_registers : boolean = (op == '1');
if t == 15 || t2 == 15 || m == 31 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VMOV \(between two general-purpose registers and two single-precision registers\)](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sm>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Vm:M" field.
<Sm1>	Is the 32-bit name of the second SIMD&FP register to be transferred. This is the next SIMD&FP register after <Sm>.
<Rt>	Is the first general-purpose register that <Sm> will be transferred to or from, encoded in the "Rt" field.
<Rt2>	Is the second general-purpose register that <Sm1> will be transferred to or from, encoded in the "Rt2" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R(t) = S(m);
        R(t2) = S(m2);
    else
        S(m) = R(t);
        S(m2) = R(t2);
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (general-purpose register to scalar)

Copy a general-purpose register to a vector element copies a byte, halfword, or word from a general-purpose register into an Advanced SIMD scalar.

On a Floating-point-only system, this instruction transfers one word to the upper or lower half of a double-precision floating-point register from a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, *HCPtr*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
!= 1111				1	1	1	0	0	opc1				0	Vd				Rt				1	0	1	1	D	opc2		1	(0)	(0)	(0)	(0)
cond																L																	

Encoding

VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if opc == '0x10' then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	1	1	0	0	opc1				0	Vd				Rt				1	0	1	1	D	opc2		1	(0)	(0)	(0)	(0)
L																																	

Encoding

VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if opc == '0x10' then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <size> The data size. It must be one of:
 - 8 Encoded as opc1<1> = 1. [x] is encoded in opc1<0>, opc2.
 - 16 Encoded as opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.
 - 32 Encoded as opc1<1> = 0, opc2 = 0b00. [x] is encoded in opc1<0>.
 - omitted Equivalent to 32.

<Dd[x]>	The scalar. The register <Dd> is encoded in D:Vd. For details of how [x] is encoded, see the description of <size>.
<Rt>	The source general-purpose register.

Operation

```

if ConditionPassed\(\) then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    D(d)[index*esize] = R(t)[esize-1:0];
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (immediate)

Copy immediate value to a SIMD&FP register places an immediate constant into every element of the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#), [A2](#), [A3](#), [A4](#) and [A5](#)) and T32 ([T1](#), [T2](#), [T3](#), [T4](#) and [T5](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd					0	x	x	0	0	Q	0	1	imm4						
cmode																										op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I32 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMOV{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	imm4H			Vd					1	0	size		(0)	0	(0)	0	imm4L			
cond																															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VMOV{<c>}{<q>}.F16 <Sd>, #<imm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let imm : bits(esize) = VFPEExpandImm{}(imm4H::imm4L);
let advsimd : boolean = FALSE;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let regs : integer = if size == '11' then 1 else 0;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd			1			0	x	0	0	Q	0	1	imm4			
																cmode				op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I16 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd			1			1	x	x	0	Q	0	1	imm4			
																cmode				op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3		Vd			1			1	1	0	0	Q	1	1	imm4			
																cmode				op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMOV{<c>}{<q>}.I64 <Qd>, #<imm>

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			0	x	x	0	0	Q	0	1	imm4				
																cmode						op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMOV{<c>}{<q>}.I32 <Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMOV{<c>}{<q>}.I32 <Qd>, #<imm>

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H			Vd			1	0	size		(0)	0	(0)	0	imm4L					

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VMOV{<c>}{<q>}.F16 <Sd>, #<imm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMOV{<c>}{<q>}.F32 <Sd>, #<imm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMOV{<c>}{<q>}.F64 <Dd>, #<imm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 8 << UInt(size);
let imm : bits(esize) = VFPEExpandImm{i}(imm4H::imm4L);
let advsimd : boolean = FALSE;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let regs : integer = if size == '11' then 1 else 0;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			1	0	x	0	0	Q	0	1	imm4				
																cmode						op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I16 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			1	1	x	x	0	Q	0	1	imm4				
																cmode						op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T5

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			1	1	1	0	0	Q	1	1	imm4				
																cmode						op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMOV{<c>}{<q>}.I64 <Qd>, #<imm>

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A3 128-bit SIMD vector", "A3 64-bit SIMD vector", "A4 128-bit SIMD vector", "A4 64-bit SIMD vector", "A5 128-bit SIMD vector", and "A5 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see [Standard assembler syntax fields](#).
For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", "T2 Single-precision scalar", "T3 128-bit SIMD vector", "T3 64-bit SIMD vector", "T4 128-bit SIMD vector", "T4 64-bit SIMD vector", "T5 128-bit SIMD vector", and "T5 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <imm> For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A3 128-bit SIMD vector", "A3 64-bit SIMD vector", "A4 128-bit SIMD vector", "A4 64-bit SIMD vector", "A5 128-bit SIMD vector", "A5 64-bit SIMD vector", "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T3 128-bit SIMD vector", "T3 64-bit SIMD vector", "T4 128-bit SIMD vector", "T4 64-bit SIMD vector", "T5 128-bit SIMD vector", and "T5 64-bit SIMD vector" variants: is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 Advanced SIMD instructions](#).
For the "A2 Double-precision scalar", "A2 Half-precision scalar", "A2 Single-precision scalar", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "imm4H:imm4L". For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 floating-point instructions](#).
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <dt> The data type, encoded in "cmode":

cmode	<dt>
110x	I32
1110	I8
1111	F32

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    if esize <= 32 then
        S(d) = ZeroExtend{32}(imm);
    else
        for r = 0 to regs-1 do
            D(d+r) = ZeroExtend{64}(imm);
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VMOV (register)

Copy between FP registers copies the contents of one FP register to another.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A2](#)) and T32 ([T2](#)) .

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	x	0	1	M	0	Vm			
cond								o1				opc2								size				o3							

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMOV{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMOV{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let single_register : boolean = (size == '10');
let advsimd : boolean = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	x	0	1	M	0	Vm			
								o1				opc2								size				o3							

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMOV{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMOV{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let single_register : boolean = (size == '10');
let advsimd : boolean = FALSE;
```

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if single_register then
         $\underline{S}(d) = \underline{S}(m);$ 
    else
         $\underline{D}(d) = \underline{D}(m);$ 
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (register, SIMD)

Copy between SIMD registers copies the contents of one SIMD register to another.

This is an alias of [VORR \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VORR \(register\)](#).
- The description of [VORR \(register\)](#) gives the operational pseudocode, any `CONSTRAINED UNPREDICTABLE` behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	0	Vn			Vd			0	0	0	1	N	Q	M	1	Vm					
U							size				opc										o1										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMOV{<c>}{<q>}{.<dt>} <Dd>, <Dm>

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>

and is the preferred disassembly when N :: Vn == M :: Vm.

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMOV{<c>}{<q>}{.<dt>} <Qd>, <Qm>

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>

and is the preferred disassembly when N :: Vn == M :: Vm.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	0	Vn			Vd			0			0	1	N	Q	M	1	Vm				
U							size				opc										o1										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMOV{<c>}{<q>}{.<dt>} <Dd>, <Dm>

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>

and is the preferred disassembly when N :: Vn == M :: Vm.

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMOV{<c>}{<q>}{.<dt>} <Qd>, <Qm>

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>

and is the preferred disassembly when N :: Vn == M :: Vm.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional.
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . See <i>Standard assembler syntax fields</i> .

<dt>	An optional data type. <dt> must not be F64, but it is otherwise ignored.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

Operation

The description of [VORR \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VORR \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (scalar to general-purpose register)

Copy a vector element to a general-purpose register with sign or zero extension copies a byte, halfword, or word from an Advanced SIMD scalar to a general-purpose register. Bytes and halfwords can be either zero-extended or sign-extended.

On a Floating-point-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, *HCPtr*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				1	1	1	0	U	opc1				1	Vn				Rt				1	0	1	1	N	opc2				1	(0)	(0)	(0)	(0)
cond																L																			

Encoding

VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x>

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if (U::opc) IN {'10x00', 'x0x10'} then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let unsigned : boolean = (U == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(N::Vn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	1	0	U	opc1				1	Vn				Rt				1	0	1	1	N	opc2				1	(0)	(0)	(0)	(0)
																L																			

Encoding

VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x>

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if (U::opc) IN {'10x00', 'x0x10'} then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let unsigned : boolean = (U == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(N::Vn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- <c> See *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.

<dt>	The data type. It must be one of:
S8	Encoded as U = 0, opc1<1> = 1. [x] is encoded in opc1<0>, opc2.
S16	Encoded as U = 0, opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.
U8	Encoded as U = 1, opc1<1> = 1. [x] is encoded in opc1<0>, opc2.
U16	Encoded as U = 1, opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.
32	Encoded as U = 0, opc1<1> = 0, opc2 = 0b00. [x] is encoded in opc1<0>.
omitted	Equivalent to 32.

<Rt>	The destination general-purpose register.
<Dn[x]>	The scalar. For details of how [x] is encoded see the description of <dt>.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if unsigned then
        R(t) = ZeroExtend{32}(D(n)[index*:esize]);
    else
        R(t) = SignExtend{32}(D(n)[index*:esize]);
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VMOVL

Vector Move Long takes each element in a doubleword vector, sign or zero-extends them to twice their original length, and places the results in a quadword vector.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm3H			0	0	0	Vd			1			0	1	0	0	0	M	1	Vm		
imm3L													opc				L		Q												

Encoding

Applies when (imm3H IN {001, 010, 100})

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

Decode for this encoding

```
if imm3H == '000' then See("Related encodings"); end;
if ! imm3H IN {'001', '010', '100'} then See("VSHLL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm3H);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let unsigned : boolean = (U == '1');
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm3H			0	0	0	Vd			1	0	1	0	0	0	M	1	Vm				
imm3L													opc				L		Q												

Encoding

Applies when (imm3H IN {001, 010, 100})

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

Decode for this encoding

```
if imm3H == '000' then See("Related encodings"); end;
if ! imm3H IN {'001', '010', '100'} then See("VSHLL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm3H);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let unsigned : boolean = (U == '1');
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the operand, encoded in “(U :: imm3H)”:

U	imm3H	<dt>
0	001	S8
0	010	S16
0	100	S32
1	001	U8
1	010	U16
1	100	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let oplelt : bits(esize) = Din(m)[e*:esize];
        let result : integer = if unsigned then UInt(oplelt) else SInt(oplelt);
        Q(d>>1)[e*(2*esize)] = result[2*esize-1:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOVN

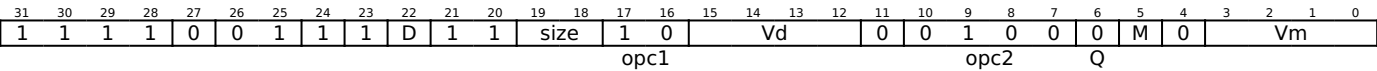
Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instructions *VRSHRN (zero)*, and *VSHRN (zero)*.

It has encodings from the following instruction sets: A32 (*A1*) and T32 (*T1*).

A1



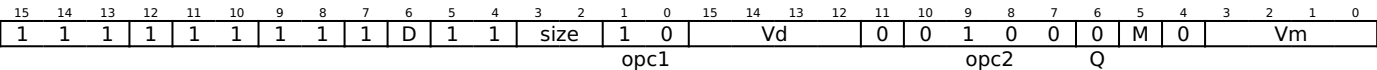
Encoding

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for this encoding

```
if size == '11' then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1



Encoding

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for this encoding

```
if size == '11' then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

- <c>For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.
- <q>See *Standard assembler syntax fields*.
- <dt>Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED
- <Dd>Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qm>Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        D(d)[e*esize] = Q(qin(m>>1)[e*(2*esize)][esize-1:0]);
    end;
end;
```


Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

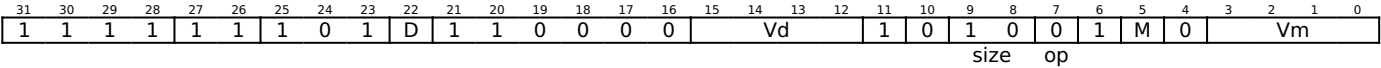
VMOVX

Vector Move extraction. This instruction copies the upper 16 bits of the 32-bit source SIMD&FP register into the lower 16 bits of the 32-bit destination SIMD&FP register, while clearing the remaining bits to zero.

Depending on settings in the *CPACR*, *NSACR*, *HCPtr*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1 (FEAT_FP16)



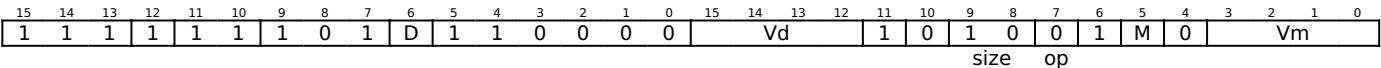
Encoding

VMOVX{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

T1 (FEAT_FP16)



Encoding

VMOVX{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = UInt(Vd::D);
let m : integer = UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

- If `InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    H(d) = S(m) [31:16];
end;
```

VMRS

Move SIMD&FP Special register to general-purpose register moves the value of an Advanced SIMD and floating-point System register to a general-purpose register. When the specified System register is the *FPSCR*, a form of the instruction transfers the *FPSCR*. {N, Z, C, V} condition flags to the *APSR*. {N, Z, C, V} condition flags.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

When these settings permit the execution of Advanced SIMD and floating-point instructions, if the specified floating-point System register is not the *FPSCR*, the instruction is UNDEFINED if executed in User mode.

In an implementation that includes EL2, when *HCR.TID0* is set to 1, any VMRS access to *FPSID* from a Non-secure EL1 mode that would be permitted if *HCR.TID0* was set to 0 generates a Hyp Trap exception. For more information, see *EL2 configurable controls*.

For simplicity, the VMRS pseudocode does not show the possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	1	1	1	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond												L																			

Encoding

VMRS{<c>} {<q>} <Rt>, <spec_reg>

Decode for this encoding

```
let t : integer = UInt(Rt);
if ! reg IN {'000x', '0101', '011x', '1000'} then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 && reg != '0001' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If ! reg IN {'000x', '0101', '011x', '1000'}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the *APSR*. {N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the Rt field, R0 - R14.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	1	1	1	1	reg				Rt				1	0	1	0	(0)	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
																L																

Encoding

VMRS{<c>} {<q>} <Rt>, <spec_reg>

Decode for this encoding

```
let t : integer = UInt(Rt);
if ! reg IN {'000x', '0101', '011x', '1000'} then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 && reg != '0001' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If ! reg IN {'000x', '0101', '011x', '1000'}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the *APSR*. {N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the Rt field, R0 - R14.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>

See *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<Rt>

Is the general-purpose destination register, encoded in the "Rt" field. Is one of:

R0-R14

General-purpose register.

APSR_nzcv

Permitted only when <spec_reg> is FPSCR. Encoded as 0b1111. The instruction transfers the *FPSCR*. {N, Z, C, V} condition flags to the *APSR*. {N, Z, C, V} condition flags.

<spec_reg>

Is the source Advanced SIMD and floating-point System register, encoded in "reg":

reg	<spec_reg>
0000	FPSID
0001	FPSCR
001x	UNPREDICTABLE
0100	UNPREDICTABLE
0101	MVFR2
0110	MVFR1
0111	MVFR0
1000	FPEXC
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then // FPSCR
        CheckVFPEEnabled(TRUE);
        if t == 15 then
            PSTATE.[N,Z,C,V] = FPSR().[N,Z,C,V];
        else
            R(t) = FPSCR();
        end;
    elseif PSTATE.EL == ELO then // Non-FPSCR registers accessible only at PL1 or above
        Undefined();
    else
        CheckVFPEEnabled(FALSE); // Non-FPSCR registers are not affected by FPEXC.EN
        AArch32.CheckAdvSIMDOrFPRegisterTraps(reg);
        case reg of
            when '0000' => R(t) = FPSID();
            when '0101' => R(t) = MVFR2();
            when '0110' => R(t) = MVFR1();
            when '0111' => R(t) = MVFR0();
            when '1000' => R(t) = FPEXC();
            otherwise => unreachable; // Dealt with above or in encoding-specific pseudocode
        end;
    end;
end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMSR

Move general-purpose register to SIMD&FP Special register moves the value of a general-purpose register to a floating-point System register. Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

When these settings permit the execution of Advanced SIMD and floating-point instructions:

- If the specified floating-point System register is [FPSID](#) or [FPEXC](#), the instruction is UNDEFINED if executed in User mode.
- If the specified floating-point System register is the [FPSID](#) and the instruction is executed in a mode other than User mode, the instruction is ignored.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	1	1	0	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond								L																							

Encoding

VMSR{<c>}{<q>} <spec_reg>, <Rt>

Decode for this encoding

```
let t : integer = UInt(Rt);
if reg != '000x' && reg != '1000' then
  let c : Constraint = ConstrainUnpredictable(Unpredictable VMSR);
  assert c IN {Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNDEF =>
      Undefined();
    when Constraint NOP =>
      ExecuteAsNOP();
  end;
end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `reg != '000x' && reg != '1000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	0	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
L																															

Encoding

VMSR{<c>}{<q>} <spec_reg>, <Rt>

Decode for this encoding

```
let t : integer = UInt(Rt);
if reg != '000x' && reg != '1000' then
  let c : Constraint = ConstrainUnpredictable(Unpredictable VMSR);
  assert c IN {Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNDEF =>
      Undefined();
    when Constraint NOP =>
      ExecuteAsNOP();
  end;
end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `reg != '000x' && reg != '1000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using `VMSR` at the same Exception level.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>See [Standard assembler syntax fields](#).

<q>See [Standard assembler syntax fields](#).

<spec_reg>Is the destination Advanced SIMD and floating-point System register, encoded in “reg”:

reg	<spec_reg>
0000	FPSID
0001	FPSCR
001x	UNPREDICTABLE
01xx	UNPREDICTABLE
1000	FPEXC
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

<Rt>Is the general-purpose source register, encoded in the "Rt" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then // FPSCR
        CheckVFPEEnabled(TRUE);
        FPSCR() = R(t);
    elseif PSTATE.EL == EL0 then // Non-FPSCR registers accessible only at PL1 or above
        Undefined();
    else // Non-FPSCR registers are not affected by FPEXC.EN
        CheckVFPEEnabled(FALSE);
        case reg of
            when '0000' => // VMSR access to FPSID is ignored
                pass;
            when '1000' =>
                FPEXC() = R(t);
            otherwise =>
                unreachable; // Dealt with above or in encoding-specific pseudocode
        end;
    end;
end;
```

VMUL (by scalar)

Vector Multiply multiplies each element in a vector by a scalar, and places the results in a second vector.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	!= 11			Vn			Vd			1 0 0			F	N	1	M	0	Vm				
size																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
  Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	Q	1	1	1	1	1	D	!= 11	Vn					Vd					1	0	0	F	N	1	M	0	Vm				
size																																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
  Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
- Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “(F :: size)”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register. When <dt> is I16 or F16, this is encoded in the "Vm<2:0>" field. Otherwise it is encoded in the "Vm" field.

<index>

Is the element index. When <dt> is I16 or F16, this is in the range 0 to 3 and is encoded in the "M:Vm<3>" field. Otherwise it is in the range 0 to 1 and is encoded in the "M" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                D(d+r)[e*:esize] = FPMul{esize}(op1elt, op2elt, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                if long_destination then
                    Q(d>>1)[e*(2*esize)] = (element1 * element2)[2*esize-1:0];
                else
                    D(d+r)[e*:esize] = (element1 * element2)[esize-1:0];
                end;
            end;
        end;
    end;
end;
```


VMUL (floating-point)

Vector Multiply multiplies corresponding elements in two vectors, and places the results in the destination vector. Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*. It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1				0	1	N	Q	M	1	Vm			
U							opc										o1																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	1	0	Vn				Vd				1	0	size	N	0	M	0	Vm				
cond				o0				o1				o2																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VMUL{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VMUL{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VMUL{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn			Vd			1	1	0	1	N	Q	M	1	Vm					
U										opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (`Q == 0`)

`VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (`Q == 1`)

`VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>`

Decode for all variants of this encoding

```
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	1	0	Vn			Vd			1	0	size	N	0	M	0	Vm						
o0								o1																o2							

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (`size == 01`)

`VMUL{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>`

Encoding for the Single-precision scalar variant

Applies when (`size == 10`)

`VMUL{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when (`size == 11`)

VMUL{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see Standard assembler syntax fields . For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table border="1"> <thead> <tr> <th>sz</th><th><dt></th></tr> </thead> <tbody> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </tbody> </table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                D(d+r)[e*:esize] = FPMul(esize)(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                H(d) = FPMul{16}(H(n), H(m), fpcr);
            when 32 =>
                S(d) = FPMul{32}(S(n), S(m), fpcr);
            when 64 =>
                D(d) = FPMul{64}(D(n), D(m), fpcr);
        end;
    end;
end;
```


VMUL (integer and polynomial)

Vector Multiply multiplies corresponding elements in two vectors.
For information about multiplying polynomials, see *Polynomial arithmetic over {0, 1}*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	op	0	D	size	Vn				Vd				1 0 0 1				N	Q	M	1				Vm			
												opc						o1															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' || (op == '1' && size != '00') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let polynomial : boolean = (op == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op	1	1	1	1	0	D	size	Vn				Vd				1	0	0	1	N	Q	M	1	Vm				
												opc						o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' || (op == '1' && size != '00') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let polynomial : boolean = (op == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the operands, encoded in "(op :: size)":

op	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	00	P8

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            var product : bits(2 * esize);
            if polynomial then
                product = PolynomialMult(esize, esize)(op1elt, op2elt);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                product = (element1 * element2)[2*esize-1:0];
            end;
            if long_destination then
                Q(d>>1)[e*(2*esize)] = product;
            else
                D(d+r)[e*:esize] = product[esize-1:0];
            end;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMULL (by scalar)

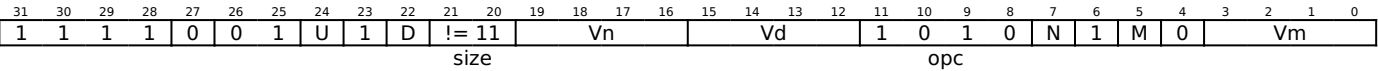
Vector Multiply Long multiplies each element in a vector by a scalar, and places the results in a second vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



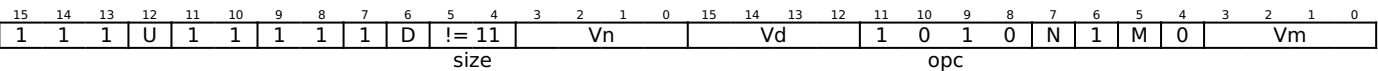
Encoding

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let floating_point : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

T1



Encoding

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let floating_point : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>

For the "T1" variant: see *Standard assembler syntax fields*.
- <dt>

See *Standard assembler syntax fields*.
- <Qd>

Is the data type for the scalar and the elements of the operand vector, encoded in “(U :: size)”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16 or U16, otherwise the "Vm" field.
<index>	Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16 or U16, otherwise in range 0 to 1, encoded in the "M" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                D(d+r)[e*:esize] = FPMul(esize)(op1elt, op2elt, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = (element1 * element2)[2*esize-1:0];
                else
                    D(d+r)[e*:esize] = (element1 * element2)[esize-1:0];
                end;
            end;
        end;
    end;
end;
end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMULL (integer and polynomial)

Vector Multiply Long multiplies corresponding elements in two vectors. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see [Polynomial arithmetic over {0, 1}](#).

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	!= 11		Vn			Vd			1 1		op	0	N	0	M	0	Vm					
size																															

Encoding

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if op == '1' && (U == '1' || size == '01') then Undefined(); end;
if op == '1' && size == '10' && !IsFeatureImplemented(FEAT_PMULL) then Undefined(); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let polynomial : boolean = (op == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = if polynomial && size == '10' then 64 else 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	!= 11			Vn			Vd			1			1	op	0	N	0	M	0	Vm		
										size																					

Encoding

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if op == '1' && (U == '1' || size == '01') then Undefined(); end;
if op == '1' && size == '10' && InITBlock() then UnpredictableProcedure(); end;
if op == '1' && size == '10' && !IsFeatureImplemented(FEAT_PMULL) then
  UnpredictableProcedure();
end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let polynomial : boolean = (op == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = if polynomial && size == '10' then 64 else 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

CONSTRAINED UNPREDICTABLE behavior

If `op == '1' && size == '10' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the operands, encoded in "(op :: U :: size)":

op	U	size	<dt>
0	0	00	S8
0	0	01	S16
0	0	10	S32
0	1	00	U8
0	1	01	U16
0	1	10	U32
1	0	00	P8
1	0	10	P64

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            var product : bits(2 * esize);
            if polynomial then
                product = PolynomialMult(esize, esize)(op1elt, op2elt);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                product = (element1 * element2)[2*esize-1:0];
            end;
            if long_destination then
                Q(d>>1)[e*(2*esize)] = product;
            else
                D(d+r)[e*:esize] = product[esize-1:0];
            end;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VMVN (immediate)

Vector Bitwise NOT (immediate) places the bitwise inverse of an immediate integer constant into every element of the destination register. Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) and [A3](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd					0	x	x	0	0	Q	1	1	imm4						
cmode																										op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd					1	0	x	0	0	Q	1	1	imm4						
cmode																										op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}.I16 <Qd>, #<imm>

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd					1	1	0	x	0	Q	1	1	imm4						
cmode																										op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0														
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			0			x			x			0			0			Q			1			1			imm4		
																cmode						op																							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			1			0			x	0	0	Q	1	1	imm4		
																cmode				op													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}.I16 <Qd>, #<imm>

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			1			1			0	x	0	Q	1	1	imm4		
																cmode				op													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", "A2 64-bit SIMD vector", "A3 128-bit SIMD vector", and "A3 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", "T2 64-bit SIMD vector", "T3 128-bit SIMD vector", and "T3 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<imm>	Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in T32 and A32 Advanced SIMD instructions .
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = NOT(imm64);
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMVN (register)

Vector Bitwise NOT (register) takes a value from a register, inverts the value of each bit, and places the result in the destination register. The registers can be either doubleword or quadword.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	0	Vd			0	1	0	1	1	Q	M	0	Vm					
opc1																opc2															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0			Vd			0	1	0	1	1	Q	M	0			Vm	
opc1																opc2															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled\(\);
    for r = 0 to regs-1 do
        D(d+r) = NOT(D(m+r));
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VNEG

Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit. Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPFXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	1	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
  Undefined();
end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	size	0	1	M	0	Vm				
cond												o1				opc2				o3											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VNEG{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNEG{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNEG{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```


CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size			0	1	Vd				0	F	1	1	1	Q	M	0	Vm		
opcl																															

Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

$$\text{VNEG}\{\langle c \rangle\} \{\langle q \rangle\} . \langle dt \rangle \langle Dd \rangle, \langle Dm \rangle$$

Encoding for the 128-bit SIMD vector variant

Applies when $(Q == 1)$

$$\text{VNEG}\{\langle c \rangle\} \{\langle q \rangle\} . \langle dt \rangle \langle Qd \rangle, \langle Qm \rangle$$

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    Undefined();
end;
if F == '1' && size == '01' && InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd					1		0	size		0	1	M	0	Vm																	
o1																opc2																o3															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VNEG{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNEG{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNEG{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see Standard assembler syntax fields . For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<dt>	Is the data type for the elements of the vectors, encoded in "(F :: size)":

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                if floating_point then
                    D(d+r)[e*esize] = FPNeg{esize}(D(m+r)[e*esize], fpcr);
                else
                    let result : integer = -SInt(D(m+r)[e*esize]);
                    D(d+r)[e*esize] = result[esize-1:0];
                end;
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 => H(d) = FPNeg{16}(H(m), fpcr);
            when 32 => S(d) = FPNeg{32}(S(m), fpcr);
            when 64 => D(d) = FPNeg{64}(D(m), fpcr);
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VNMLA

Vector Negate Multiply Accumulate multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

Note

Arm recommends that software does not use the VNMLA instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	0	1	Vn				Vd				1	0	size		N	1	M	0	Vm			
cond				o0				o1												op											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	0	1	Vn				Vd				1	0	size		N	1	M	0	Vm			
o0								o1				op																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let product16 : bits(16) = FPMul{ } (H(n), H(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => H(d) = FPAAdd{16} (FPNeg{16} (H(d), fpcr),
                                                                FPNeg{16} (product16, fpcr), fpcr);
                when VFPNegMul_VNMLS => H(d) = FPAAdd{16} (FPNeg{16} (H(d), fpcr), product16, fpcr);
                when VFPNegMul_VNMUL => H(d) = FPNeg{16} (product16, fpcr);
            end;
        when 32 =>
            let product32 : bits(32) = FPMul{ } (S(n), S(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => S(d) = FPAAdd{32} (FPNeg{32} (S(d), fpcr),
                                                                FPNeg{32} (product32, fpcr), fpcr);
                when VFPNegMul_VNMLS => S(d) = FPAAdd{32} (FPNeg{32} (S(d), fpcr), product32, fpcr);
                when VFPNegMul_VNMUL => S(d) = FPNeg{32} (product32, fpcr);
            end;
        when 64 =>
            let product64 : bits(64) = FPMul{ } (D(n), D(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => D(d) = FPAAdd{64} (FPNeg{64} (D(d), fpcr),
                                                                FPNeg{64} (product64, fpcr), fpcr);
                when VFPNegMul_VNMLS => D(d) = FPAAdd{64} (FPNeg{64} (D(d), fpcr), product64, fpcr);
                when VFPNegMul_VNMUL => D(d) = FPNeg{64} (product64, fpcr);
            end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VNMLS

Vector Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	0	1	Vn				Vd				1 0		size	N	0	M	0	Vm				
cond				o0				o1																op							

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	1	0	0	D	0	1	Vn				Vd				1		0	size		N	0	M	0	Vm						
								o0				o1																op							

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let product16 : bits(16) = FPMul{ }(H(n), H(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => H(d) = FPAdd{16}(FPNeg{16}(H(d), fpcr),
                                                            FPNeg{16}(product16, fpcr), fpcr);
                when VFPNegMul_VNMLS => H(d) = FPAdd{16}(FPNeg{16}(H(d), fpcr), product16, fpcr);
                when VFPNegMul_VNMUL => H(d) = FPNeg{16}(product16, fpcr);
            end;
        when 32 =>
            let product32 : bits(32) = FPMul{ }(S(n), S(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr),
                                                            FPNeg{32}(product32, fpcr), fpcr);
                when VFPNegMul_VNMLS => S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr), product32, fpcr);
                when VFPNegMul_VNMUL => S(d) = FPNeg{32}(product32, fpcr);
            end;
        when 64 =>
            let product64 : bits(64) = FPMul{ }(D(n), D(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr),
                                                            FPNeg{64}(product64, fpcr), fpcr);
                when VFPNegMul_VNMLS => D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr), product64, fpcr);
                when VFPNegMul_VNMUL => D(d) = FPNeg{64}(product64, fpcr);
            end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VNMUL

Vector Negate Multiply multiplies together two floating-point register values, and writes the negation of the result to the destination register. Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	1	0	Vn				Vd				1	0	size	N	1	M	0	Vm				
cond				o0				o1				o2																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VNMUL{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMUL{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMUL{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let vtype : VFPNegMul = VFPNegMul VNMUL;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	1	1	0	0	D	1	0	Vn				Vd				1	0	size	N	1	M	0	Vm						
								o0				o1				o2																	

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VNMUL{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMUL{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)


```
VNMUL{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let vtype : VFPNegMul = VFPNegMul_VNMUL;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 =>
            let product16 : bits(16) = FPMul{ } (H(n), H(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => H(d) = FPAdd{16} (FPNeg{16} (H(d), fpcr),
                                                            FPNeg{16} (product16, fpcr), fpcr);
                when VFPNegMul_VNMLS => H(d) = FPAdd{16} (FPNeg{16} (H(d), fpcr), product16, fpcr);
                when VFPNegMul_VNMUL => H(d) = FPNeg{16} (product16, fpcr);
            end;
        when 32 =>
            let product32 : bits(32) = FPMul{ } (S(n), S(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => S(d) = FPAdd{32} (FPNeg{32} (S(d), fpcr),
                                                            FPNeg{32} (product32, fpcr), fpcr);
                when VFPNegMul_VNMLS => S(d) = FPAdd{32} (FPNeg{32} (S(d), fpcr), product32, fpcr);
                when VFPNegMul_VNMUL => S(d) = FPNeg{32} (product32, fpcr);
            end;
        when 64 =>
            let product64 : bits(64) = FPMul{ } (D(n), D(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => D(d) = FPAdd{64} (FPNeg{64} (D(d), fpcr),
                                                            FPNeg{64} (product64, fpcr), fpcr);
                when VFPNegMul_VNMLS => D(d) = FPAdd{64} (FPNeg{64} (D(d), fpcr), product64, fpcr);
                when VFPNegMul_VNMUL => D(d) = FPNeg{64} (product64, fpcr);
            end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VORN (immediate)

Vector Bitwise OR NOT (immediate) performs a bitwise OR between a register value and the complement of an immediate value, and returns the result into the destination vector.

This is a pseudo-instruction of [VORR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [VORR \(immediate\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VORR \(immediate\)](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd					0	x	x	1	0	Q	0	1	imm4						
cmode																										op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$\text{VORN}\{\langle c \rangle\}\{\langle q \rangle\}.I32\ \{\langle Dd \rangle,\ \}\langle Dd \rangle,\ \#\langle imm \rangle$

is equivalent to

$\text{VORR}\{\langle c \rangle\}\{\langle q \rangle\}.I32\ \langle Dd \rangle,\ \#\sim\langle imm \rangle$

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

$\text{VORN}\{\langle c \rangle\}\{\langle q \rangle\}.I32\ \{\langle Qd \rangle,\ \}\langle Qd \rangle,\ \#\langle imm \rangle$

is equivalent to

$\text{VORR}\{\langle c \rangle\}\{\langle q \rangle\}.I32\ \langle Qd \rangle,\ \#\sim\langle imm \rangle$

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd					1	0	x	1	0	Q	0	1	imm4						
cmode																										op									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$\text{VORN}\{\langle c \rangle\}\{\langle q \rangle\}.I16\ \{\langle Dd \rangle,\ \}\langle Dd \rangle,\ \#\langle imm \rangle$

is equivalent to

$\text{VORR}\{\langle c \rangle\}\{\langle q \rangle\}.I16\ \langle Dd \rangle,\ \#\sim\langle imm \rangle$

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

$\text{VORN}\{\langle c \rangle\}\{\langle q \rangle\}.I16\ \{\langle Qd \rangle,\ \}\langle Qd \rangle,\ \#\langle imm \rangle$

is equivalent to

$\text{VORR}\{\langle c \rangle\}\{\langle q \rangle\}.I16\ \langle Qd \rangle,\ \#\sim\langle imm \rangle$

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd					0	x	x	1	0	Q	0	1	imm4													
																										cmode					op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

$\text{VORN}\{\langle c \rangle\}\{\langle q \rangle\}.I32\ \{\langle Dd \rangle,\ \}\langle Dd \rangle,\ \#\langle imm \rangle$

is equivalent to

VORR{<c>}{<q>}.I32 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

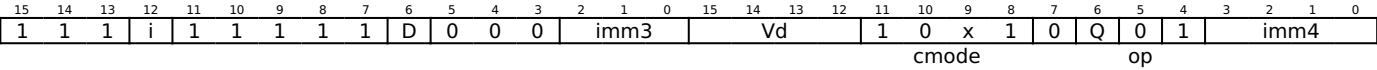
Applies when (Q == 1)

VORN{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I32 <Qd>, #~<imm>

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I16 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORN{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I16 <Qd>, #~<imm>

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <q>

See *Standard assembler syntax fields*.
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions*.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Operation

The description of [VORR \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VORR \(immediate\)](#) gives the operational information for this instruction.

VORN (register)

Vector bitwise OR NOT (register) performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	1	Vn			Vd			0			0	0	1	N	Q	M	1	Vm			
U							size				opc										o1										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORN{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	1	Vn			Vd			0	0	0	1	N	Q	M	1	Vm					
U							size				opc										o1										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORN{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled\(\);
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) OR NOT(D(m+r));
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VORR (immediate)

Vector Bitwise OR (immediate) performs a bitwise OR between a register value and an immediate value, and returns the result into the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VORN \(immediate\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd			0	x	x	1	0	Q	0	1	imm4				
																cmode				op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VORR{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VORR{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd			1	0	x	1	0	Q	0	1	imm4				
																cmode				op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VORR{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VORR{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			0	x	x	1	0	Q	0	1	imm4				
																cmode				op											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VORR{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORR{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

Decode for all variants of this encoding

```

if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;

```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			1			0			x	1	0	Q		0	1	imm4		
																cmode								op										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORR{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORR{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

Decode for all variants of this encoding

```

if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;

```

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <Dd> See *Standard assembler syntax fields*.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <imm> Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions*.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(d+r) OR imm64;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VORR (register)

Vector bitwise OR (register) performs a bitwise OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the alias *VMOV (register, SIMD)*.

This instruction is used by the pseudo-instructions *VRSHR (zero)*, and *VSHR (zero)*.

It has encodings from the following instruction sets: A32 (*A1*) and T32 (*T1*).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	0	1	0	0	D	1	0	Vn				Vd				0				0	0	1	N	Q	M	1	Vm			
U							size				opc										ol													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			
U							size				opc										ol										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Alias Conditions

Alias	Is preferred when
VMOV (register, SIMD)	N :: Vn == M :: Vm
VRSHR (zero)	Never
VSHR (zero)	Never

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) OR D(m+r);
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

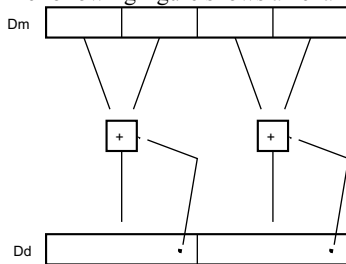
Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPADAL

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows an example of the operation of VPADAL doubleword operation for data type S16.



Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	0	Vd			0			1	1	0	op	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0		Vd				0	1	1	0	op	Q	M	0			Vm	
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

See *Standard assembler syntax fields*.

Is the data type for the elements of the vectors, encoded in “(op :: size)”:

op	size	<dt>
0	00	S8
0	01	S16
0	10	S32
x	11	RESERVED
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let h : integer = elements DIV 2;

    for r = 0 to regs-1 do
        for e = 0 to h-1 do
            let op1elt : bits(esize) = D(m+r)[(2*e)*:esize];
            let op2elt : bits(esize) = D(m+r)[(2*e+1)*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = element1 + element2;
            D(d+r)[e*(2*esize)] = D(d+r)[e*(2*esize)] + result;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VPADD (floating-point)

Vector Pairwise Add (floating-point) adds adjacent pairs of elements of two vectors, and places the results in the destination vector. The operands and result are doubleword vectors. The operand and result elements are floating-point numbers. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support. It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
U												opc								o1											

Encoding

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if Q == '1' then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
U												opc								o1											

Encoding

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if Q == '1' then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

CONSTRAINED UNPREDICTABLE behavior

- If sz == '1' && InITBlock(), then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1" variant: see Standard assembler syntax fields. This encoding must be unconditional. For the "T1" variant: see Standard assembler syntax fields.						
<q>	See Standard assembler syntax fields.						
<dt>	Is the data type for the elements of the vectors, encoded in “sz”: <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    var dest : bits(64);
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        dest[e*:esize] = FPAdd{esize}( D(n)[(2*e)*:esize], D(n)[(2*e+1)*:esize], fpcr);
        dest[(e+h)*:esize] = FPAdd{esize}( D(m)[(2*e)*:esize], D(m)[(2*e+1)*:esize], fpcr);
    end;

    D(d) = dest;
end;
```

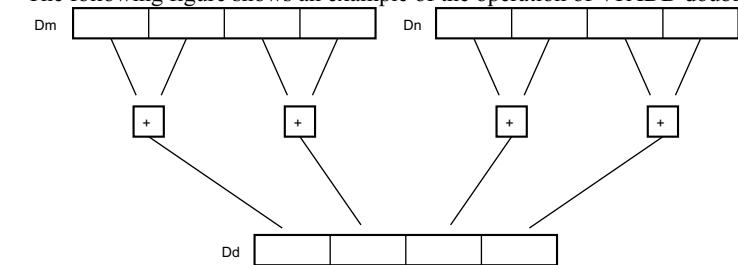
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector. The operands and result are doubleword vectors. The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The following figure shows an example of the operation of VPADD doubleword operation for data type I16.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size					Vn				Vd		1	0	1	1	N	Q	M	1			Vm
U										opc						o1															

Encoding

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if size == '11' || Q == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size					Vn				Vd		1	0	1	1	N	Q	M	1		Vm	
U										opc						o1															

Encoding

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if size == '11' || Q == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

- <c>For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.
- <q>See *Standard assembler syntax fields*.
- <dt>Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
- <Dd>Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn>Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        dest[e*:esize] = D(n)[(2*e)*:esize] + D(n)[(2*e+1)*:esize];
        dest[(e+h)*:esize] = D(m)[(2*e)*:esize] + D(m)[(2*e+1)*:esize];
    end;

    D(d) = dest;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

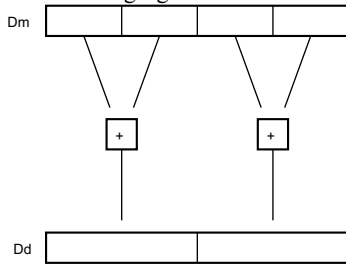
Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPADDL

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows an example of the operation of VPADDL doubleword operation for data type S16.



Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd			0		0	1	0	op	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0					Vd	0	0	1	0	op	Q	M	0			Vm	
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the vectors, encoded in “(op :: size)”:
- | op | size | <dt> |
|----|------|----------|
| 0 | 00 | S8 |
| 0 | 01 | S16 |
| 0 | 10 | S32 |
| x | 11 | RESERVED |
| 1 | 00 | U8 |
| 1 | 01 | U16 |
| 1 | 10 | U32 |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let h : integer = elements DIV 2;

    for r = 0 to regs-1 do
        for e = 0 to h-1 do
            let op1elt : bits(esize) = D(m+r)[(2*e)*:esize];
            let op2elt : bits(esize) = D(m+r)[(2*e+1)*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = element1 + element2;
            D(d+r)[e*(2*esize)] = result[2*esize-1:0];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

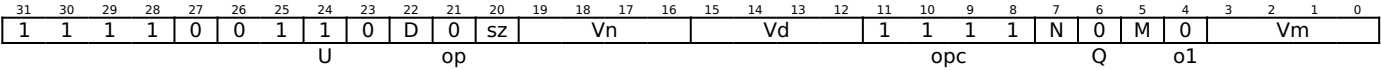
VPMAX (floating-point)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

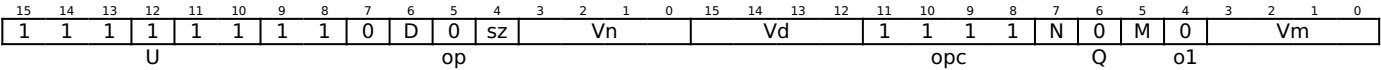
Decode for this encoding

```

if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1



Encoding

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```

if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

CONSTRAINED UNPREDICTABLE behavior

- If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1" variant: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1" variant: see <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the vectors, encoded in “sz”: <table border="1"> <thead> <tr> <th>sz</th><th><dt></th></tr> </thead> <tbody> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </tbody> </table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let fpcr : FPCR_Type = StandardFPCR();
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        dest[e*:esize] = (if maximum then FPMax{esize}(op1elt, op2elt, fpcr)
                        else FPMin{esize}(op1elt, op2elt, fpcr));
        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        dest[(e+h)*:esize] = (if maximum then FPMax{esize}(op1elt, op2elt, fpcr)
                        else FPMin{esize}(op1elt, op2elt, fpcr));
    end;

    D(d) = dest;
end;
```

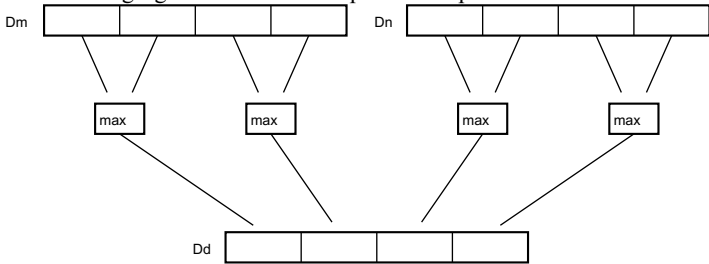
2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPMAX (integer)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

The following figure shows an example of the operation of VPMAX doubleword operation for data type S16 or U16.



Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn				Vd				1	0	1	0	N	0	M	0	Vm				
											opc				Q				op												

Encoding

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size	Vn				Vd				1	0	1	0	N	0	M	0	Vm				
											opc				Q				op												

Encoding

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

- <c> For the "A1" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1" variant: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).

<dt> Is the data type for the elements of the operands, encoded in “(U :: size)”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        var element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        var element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        var result : integer = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[e*:esize] = result[esize-1:0];
        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        element2 = if unsigned then UInt(op2elt) else SInt(op2elt);
        result = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[(e+h)*:esize] = result[esize-1:0];
    end;

    D(d) = dest;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

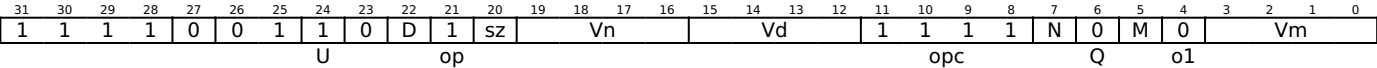
VPMIN (floating-point)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



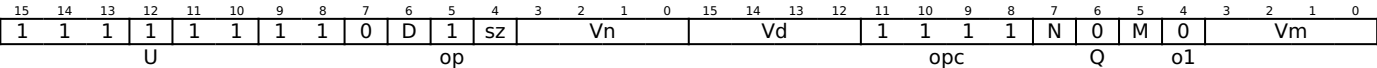
Encoding

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1



Encoding

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

CONSTRAINED UNPREDICTABLE behavior

- If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
 - The instruction executes as if it passes the Condition code check.
 - The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1" variant: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For the "T1" variant: see <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the vectors, encoded in “sz”: <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let fpcr : FPCR_Type = StandardFPCR();
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        dest[e*:esize] = (if maximum then FPMax{esize}(op1elt, op2elt, fpcr)
                        else FPMin{esize}(op1elt, op2elt, fpcr));
        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        dest[(e+h)*:esize] = (if maximum then FPMax{esize}(op1elt, op2elt, fpcr)
                            else FPMin{esize}(op1elt, op2elt, fpcr));
    end;

    D(d) = dest;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

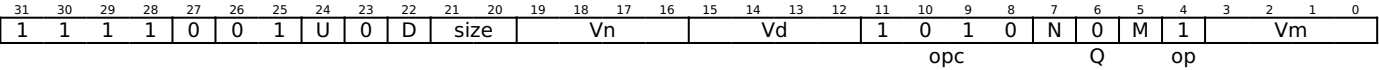
VPMIN (integer)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



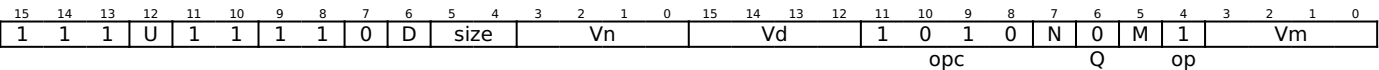
Encoding

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1



Encoding

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

- <c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>

For the "T1" variant: see *Standard assembler syntax fields*.
- <dt>

See *Standard assembler syntax fields*.
- Is the data type for the elements of the operands, encoded in “(U :: size)”:
- | U | size | <dt> |
|---|------|------|
| 0 | 00 | S8 |
| 0 | 01 | S16 |
| 0 | 10 | S32 |
| 1 | 00 | U8 |
| 1 | 01 | U16 |
| 1 | 10 | U32 |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        var element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        var element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        var result : integer = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[e*:esize] = result[esize-1:0];
        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        element2 = if unsigned then UInt(op2elt) else SInt(op2elt);
        result = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[(e+h)*:esize] = result[esize-1:0];
    end;

    D(d) = dest;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPOP

Pop SIMD&FP registers from stack loads multiple consecutive Advanced SIMD and floating-point register file registers from the stack.

This is an alias of [VLDM](#), [VLDMDB](#), [VLDMIA](#). This means:

- The encodings in this description are named to match the encodings of [VLDM](#), [VLDMDB](#), [VLDMIA](#).
- The description of [VLDM](#), [VLDMDB](#), [VLDMIA](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	1	x	x	x	x	x	x	x	0
cond				P		U	W		L	Rn				size				imm8													

Encoding for the Increment After variant

VPOP{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

[VLDM](#){<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	0	imm8							
cond				P		U	W		L	Rn				size																	

Encoding for the Increment After variant

VPOP{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

[VLDM](#){<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	1	x	x	x	x	x	x	x	0
				P		U	W		L	Rn				size				imm8													

Encoding for the Increment After variant

VPOP{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

[VLDM](#){<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	0	imm8							
				P		U	W		L	Rn				size																	

Encoding for the Increment After variant

VPOP{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

[VLDM](#){<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<size>	An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
<sreglist>	Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
<dreglist>	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Operation

The description of [VLDM](#), [VLDMDB](#), [VLDMIA](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VLDM](#), [VLDMDB](#), [VLDMIA](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPUSH

Push SIMD&FP registers to stack stores multiple consecutive registers from the Advanced SIMD and floating-point register file to the stack.

This is an alias of [VSTM](#), [VSTMDB](#), [VSTMIA](#). This means:

- The encodings in this description are named to match the encodings of [VSTM](#), [VSTMDB](#), [VSTMIA](#).
- The description of [VSTM](#), [VSTMDB](#), [VSTMIA](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	1	1	x	x	x	x	x	x	0
cond				P		U	W		L	Rn				size				imm8													

Encoding for the Decrement Before variant

`VPUSH{<c>}{<q>}{.<size>} <dreglist>`

is equivalent to

`VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>`

and is always the preferred disassembly.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	0	imm8											
cond				P		U	W		L	Rn				size																					

Encoding for the Decrement Before variant

`VPUSH{<c>}{<q>}{.<size>} <sreglist>`

is equivalent to

`VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>`

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	1	x	x	x	x	x	x	x	0
				P		U	W		L	Rn				size				imm8													

Encoding for the Decrement Before variant

`VPUSH{<c>}{<q>}{.<size>} <dreglist>`

is equivalent to

`VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>`

and is always the preferred disassembly.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	0	D	1	0	1	0	1	0	Vd				1	0	1	0	imm8							
				P		U	W		L	Rn				size																	

Encoding for the Decrement Before variant

`VPUSH{<c>}{<q>}{.<size>} <sreglist>`

is equivalent to

`VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>`

and is always the preferred disassembly.

Assembler Symbols

<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<size>	An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
<sreglist>	Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
<dreglist>	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Operation

The description of [VSTM](#), [VSTMDB](#), [VSTMIA](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VSTM](#), [VSTMDB](#), [VSTMIA](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

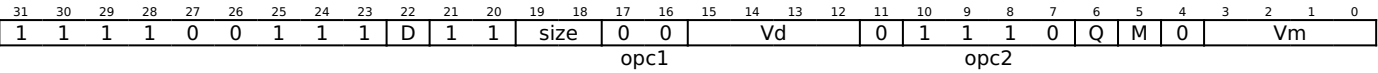
VQABS

Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector. If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR.QC*, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

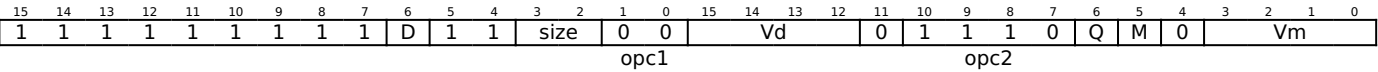
Applies when (Q == 1)

```
VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c>
- For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>
- For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let result : integer = Abs(SInt(D(m+r)[e*:esize]));
            let (res, sat) : (bits(esize), boolean) = SignedSatQ{esize}(result);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

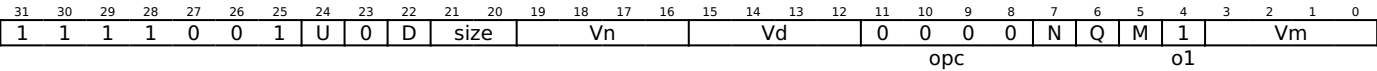
VQADD

Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector. If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR.QC*, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

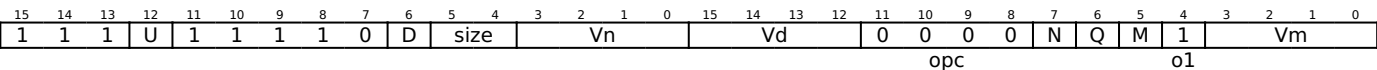
Applies when (Q == 1)

VQADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “(U :: size)”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let sum : integer = element1 + element2;
            let (res, sat) : (bits(esize), boolean) = SatQ(esize){sum, unsigned};
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```

VQDMLAL

Vector Saturating Doubling Multiply Accumulate Long multiplies corresponding elements in two doubleword vectors, doubles the products, and accumulates the results into the elements of a quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	!= 11			Vn			Vd			1	0	0	1	N	0	M	0	Vm				
U										size			op																		

Encoding

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	!= 11			Vn			Vd			0	0	1	1	N	1	M	0	Vm				
Q										size			op																		

Encoding

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	!= 11			Vn			Vd			1	0	0	1	N	0	M	0	Vm				
U										size			op																		

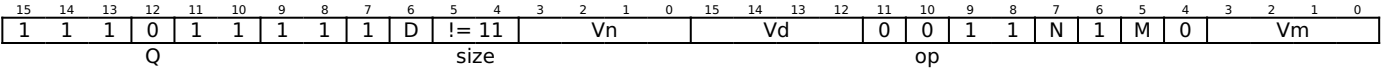
Encoding

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

T2



Encoding

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" and "A2" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1" and "T2" variants: see [Standard assembler syntax fields](#).
- <q>

See [Standard assembler syntax fields](#).
- <dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>

For the "A1" and "T1" variants: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

For the "A2" and "T2" variants: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.
- <index>

Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(Din(m)[index*:esize]); end;
    for e = 0 to elements-1 do
        if !scalar_form then op2 = SInt(Din(m)[e*:esize]); end;
        let op1 : integer = SInt(Din(n)[e*:esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        let (product, sat1) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(2*op1*op2);
        var result : integer;
        if add then
            result = SInt(Qin(d>>1)[e*(2*esize)]) + SInt(product);
        else
            result = SInt(Qin(d>>1)[e*(2*esize)]) - SInt(product);
        end;
        var sat2 : boolean;
        var res : bits(2*esize);
        (res, sat2) = SignedSatQ{2*esize}(result);
        Q(d>>1)[e*(2*esize)] = res;
        if sat1 || sat2 then FPSCR().QC = '1'; end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQDMLSL

Vector Saturating Doubling Multiply Subtract Long multiplies corresponding elements in two doubleword vectors, subtracts double the products from corresponding elements of a quadword vector, and places the results in the same quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	!= 11			Vn			Vd			1 0			1	1	N	0	M	0	Vm			
U										size			op																		

Encoding

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	!= 11			Vn			Vd			0	1	1	1	N	1	M	0	Vm				
Q										size			op																		

Encoding

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	!= 11			Vn			Vd			1	0	1	1	N	0	M	0	Vm				
U										size			op																		

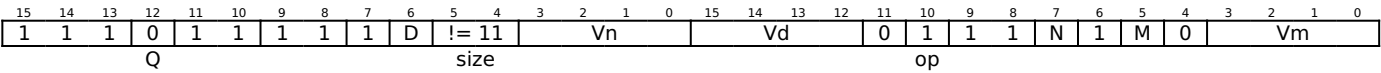
Encoding

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

T2



Encoding

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

- <c>
For the "A1" and "A2" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1" and "T2" variants: see [Standard assembler syntax fields](#).
- <q>
See [Standard assembler syntax fields](#).
- <dt>
Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32
- <Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>
For the "A1" and "T1" variants: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
For the "A2" and "T2" variants: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.
- <index>
Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(Din(m)[index*:esize]); end;
    for e = 0 to elements-1 do
        if !scalar_form then op2 = SInt(Din(m)[e*:esize]); end;
        let op1 : integer = SInt(Din(n)[e*:esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        let (product, sat1) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(2*op1*op2);
        var result : integer;
        if add then
            result = SInt(Qin(d>>1)[e*(2*esize)]) + SInt(product);
        else
            result = SInt(Qin(d>>1)[e*(2*esize)]) - SInt(product);
        end;
        let (res, sat2) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(result);
        Q(d>>1)[e*(2*esize)] = res;
        if sat1 || sat2 then FPSCR().QC = '1'; end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated, for rounded results see [VQRDMULH](#).

The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size	Vn				Vd				1	0	1	1	N	Q	M	0	Vm				
U										opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	Q	1	D	!= 11	Vn					Vd					1	1	0	0	N	1	M	0	Vm			
size											opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm[x]>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size	Vn			Vd			1	0	1	1	N	Q	M	0	Vm						
U										opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

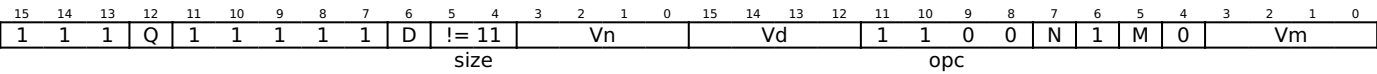
Applies when (Q == 1)

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm[x>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt> See [Standard assembler syntax fields](#).
- Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(D(m)[index*:esize]); end;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if !scalar_form then op2 = SInt(D(m+r)[e*:esize]); end;
            let op1 : integer = SInt(D(n+r)[e*:esize]);
            // The following only saturates if both op1 and op2 equal -(2^(esize-1))
            let (result, sat) : (bits(esize), boolean) = SignedSatQ{esize}((2*op1*op2) >> esize);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQDMULL

Vector Saturating Doubling Multiply Long multiplies corresponding elements in two doubleword vectors, doubles the products, and places the results in a quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	!= 11		Vn				Vd				1		1	0	1	N	0	M	0	Vm		
U										size										opc											

Encoding

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	!= 11		Vn				Vd				1	0	1	1	N	1	M	0	Vm			
Q										size										opc											

Encoding

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
1	1	1	0	1	1	1	1	1	D	!= 11						Vn					Vd					1	1	0	1	N	0	M	0	Vm				
U										size										opc																		

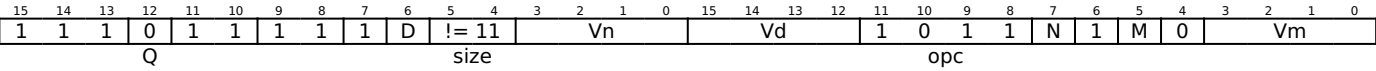
Encoding

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;
```

T2



Encoding

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

- <c> For the "A1" and "A2" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q> For the "T1" and "T2" variants: see [Standard assembler syntax fields](#).
- <dt> Is the data type for the elements of the operands, encoded in "size":

size	<dt>
01	S16
10	S32
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(Din(m)[index*esize]); end;
    for e = 0 to elements-1 do
        if !scalar_form then op2 = SInt(Din(m)[e*esize]); end;
        let op1 : integer = SInt(Din(n)[e*esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        let (product, sat) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(2*op1*op2);
        Q(d>>1)[e*(2*esize)] = product;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```

VQMOVN, VQMOVUN

Vector Saturating Move and Narrow copies each element of the operand vector to the corresponding element of the destination vector. The operand is a quadword vector. The elements can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result is a doubleword vector. The elements are half the length of the operand vector elements. If the operand is unsigned, the results are unsigned. If the operand is signed, the results can be signed or unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instructions [VQRSHRN \(zero\)](#), [VQRSHRUN \(zero\)](#), [VQSHRN \(zero\)](#), and [VQSHRUN \(zero\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	op		M	0	Vm			

opc1

Encoding for the Signed result variant

Applies when (op == 1x)

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Encoding for the Unsigned result variant

Applies when (op == 01)

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for all variants of this encoding

```
if op == '00' then See("VMOVN"); end;
if size == '11' || Vm[0] == '1' then Undefined(); end;
let src_unsigned : boolean = (op == '11');
let dest_unsigned : boolean = (op[0] == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0					Vd		0	0	1	0	op	M	0			Vm	

opc1

Encoding for the Signed result variant

Applies when (op == 1x)

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Encoding for the Unsigned result variant

Applies when (op == 01)

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for all variants of this encoding

```
if op == '00' then See("VMOVN"); end;
if size == '11' || Vm[0] == '1' then Undefined(); end;
let src_unsigned : boolean = (op == '11');
let dest_unsigned : boolean = (op[0] == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c> For the "A1 Signed result" and "A1 Unsigned result" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1 Signed result" and "T1 Unsigned result" variants: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> For the "A1 Signed result" and "T1 Signed result" variants: is the data type for the elements of the operand, encoded in “(op :: size)”:

op	size	<dt>
10	00	S16
10	01	S32
10	10	S64
1x	11	RESERVED
11	00	U16
11	01	U32
11	10	U64

For the "A1 Unsigned result" and "T1 Unsigned result" variants: is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

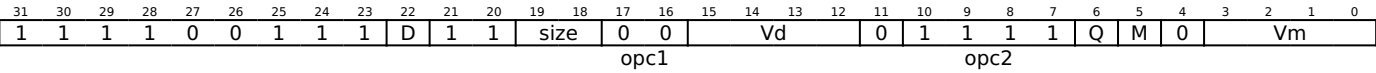
Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  for e = 0 to elements-1 do
    let opelt : bits(2*esize) = Qin(m>>1)[e:(2*esize)];
    let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
    let (res, sat) : (bits(esize), boolean) = SatQ(esize)(element, dest_unsigned);
    D(d)[e*:esize] = res;
    if sat then FPSCR().QC = '1'; end;
  end;
end;
```

VQNEG

Vector Saturating Negate negates each element in a vector, and places the results in the destination vector. If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR.QC*, is set if saturation occurs. For details see *Pseudocode details of saturation*. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

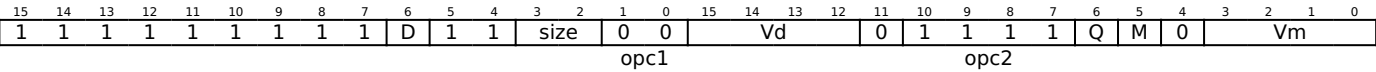
Applies when (Q == 1)

VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  for r = 0 to regs-1 do
    for e = 0 to elements-1 do
      let result : integer = -SInt(D(m+r)[e*:esize]);
      let (res, sat) : (bits(esize), boolean) = SignedSatQ{esize}(result);
      D(d+r)[e*:esize] = res;
      if sat then FPSCR().QC = '1'; end;
    end;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRDMLAH

Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR.QC*, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPT* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1				0	1	1	N	Q	M	1	Vm			
U										opc										o1														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let add : boolean = TRUE;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

A2

(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	!= 11		Vn				Vd				1	1	1	0	N	1	M	0	Vm			
size										opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[x>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let add : boolean = TRUE;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

(FEAT_RDM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	size	Vn		Vd		1	0	1	1	N	Q	M	1	Vm								
U										opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let add : boolean = TRUE;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

(FEAT_RDM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	!= 11	Vn		Vd		1	1	1	0	N	1	M	0	Vm								
size										opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let add : boolean = TRUE;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<q>	See Standard assembler syntax fields .
<dt>	Is the data type for the elements of the operands, encoded in "size":
size	<dt>
01	S16
10	S32
<Dd>	Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dm[x]>	Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
var operand2 : integer;
let round : boolean = TRUE;
if scalar_form then operand2 = SInt(D(m)[index*:esize]); end;
for r = 0 to regs-1 do
  for e = 0 to elements-1 do
    let operand1 : integer = SInt(D(n+r)[e*:esize]);
    let operand3 : integer = SInt(D(d+r)[e*:esize]) << esize;
    if !scalar_form then operand2 = SInt(D(m+r)[e*:esize]); end;
    let rdmlah : integer = RShr(operand3 + 2*(operand1*operand2), esize, round);
    let (result, sat) : (bits(esize), boolean) = SignedSatQ{esize}(rdmlah);
    D(d+r)[e*:esize] = result;
    if sat then FPSCR().QC = '1'; end;
  end;
end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRDMLSH

Vector Saturating Rounding Doubling Multiply Subtract Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR.QC*, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1		1	0	0	N	Q	M	1	Vm			
U										opc										o1												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMLSH(<q>).<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMLSH(<q>).<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let add : boolean = FALSE;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

A2

(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	!= 11		Vn				Vd				1	1	1	1	N	1	M	0	Vm			
size										opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMLSH(<q>).<dt> <Dd>, <Dn>, <Dm[x>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMLSH(<q>).<dt> <Qd>, <Qn>, <Dm[x>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let add : boolean = FALSE;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

(FEAT_RDM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	1	1	0	D	size	Vn					Vd					1	1	0	0	N	Q	M	1	Vm				
U										opc										o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let add : boolean = FALSE;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

(FEAT_RDM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	!= 11			Vn			Vd			1			1	1	1	N	1	M	0	Vm		
size										opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then Undefined(); end;
if InITBlock() then UnpredictableProcedure(); end;
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let add : boolean = FALSE;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<q>	See Standard assembler syntax fields .
<dt>	Is the data type for the elements of the operands, encoded in "size":
size	<dt>
01	S16
10	S32
<Dd>	Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dm[x]>	Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```

EncodingSpecificOperations();
CheckAdvSIMDEnabled();
var operand2 : integer;
let round : boolean = TRUE;
if scalar_form then operand2 = SInt(D(m)[index*:esize]); end;
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        let operand1 : integer = SInt(D(n+r)[e*:esize]);
        let operand3 : integer = SInt(D(d+r)[e*:esize]) << esize;
        if !scalar_form then operand2 = SInt(D(m+r)[e*:esize]); end;
        let rdmlsh : integer = RShr(operand3 - 2*(operand1*operand2), esize, round);
        let (result, sat) : (bits(esize), boolean) = SignedSatQ{esize}(rdmlsh);
        D(d+r)[e*:esize] = result;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRDMULH

Vector Saturating Rounding Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are rounded. For truncated results see [VQDMULH](#).

The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size	Vn			Vd			1	0	1	1	N	Q	M	0	Vm						
U										opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	!= 11	Vn				Vd				1	1	0	1	N	1	M	0	Vm				
size											opc																				

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm[x]>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	size	Vn			Vd			1	0	1	1	N	Q	M	0	Vm						
U										opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

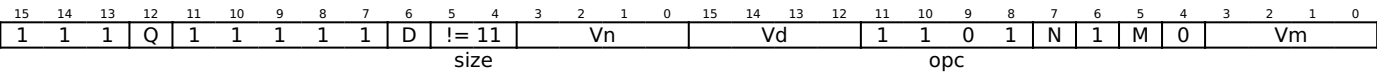
Applies when (Q == 1)

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;
```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x]>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector", "A1 64-bit SIMD vector", "A2 128-bit SIMD vector", and "A2 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 128-bit SIMD vector", and "T2 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt> See [Standard assembler syntax fields](#).
- Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  var op2 : integer;
  let round : boolean = TRUE;
  if scalar_form then op2 = SInt(D(m)[index*:esize]); end;
  for r = 0 to regs-1 do
    for e = 0 to elements-1 do
      let op1 : integer = SInt(D(n+r)[e*:esize]);
      if !scalar_form then op2 = SInt(D(m+r)[e*:esize]); end;
      let rdmulh : integer = RShr(2*op1*op2, esize, round);
      let (result, sat) : (bits(esize), boolean) = SignedSatQ{esize}(rdmulh);
      D(d+r)[e*:esize] = result;
      if sat then FPSCR().QC = '1'; end;
    end;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see [VQSHL \(register\)](#).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	1	1	0	0	1	U	0	D	size	Vn				Vd				0	1	0	1	N	Q	M	1	Vm												
																				opc				o1															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	0	D	size	Vn				Vd				0				1	0	1	N	Q	M	1	Vm			
												opc																o1					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the vectors, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var result : bits(esize);
    var sat : boolean;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            var element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            if shift >= 0 then // left shift
                element = element << shift;
            else // rounding right shift
                shift = -shift;
                element = (element + (1 << (shift - 1))) >> shift;
            end;
            (result, sat) = SatQ{esize}(element, unsigned);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```

VQRSHRN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed rounded results in a doubleword vector.

This is a pseudo-instruction of [VQMOVN, VQMOVUN](#). This means:

- The encodings in this description are named to match the encodings of [VQMOVN, VQMOVUN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VQMOVN, VQMOVUN](#) gives the operational pseudocode, any CONstrained UNpredictable behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	1	D	1	1	size			1	0	Vd					0	0	1	0	1		x	M	0	Vm		
opc1														op																			

Encoding for the Signed result variant

VQRSHRN{<c>} {<q>} .<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>} {<q>} .<dt> <Dd>, <Qm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd				0	0	1	0	1	x	M	0	Vm				
opc1														op																	

Encoding for the Signed result variant

VQRSHRN{<c>} {<q>} .<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>} {<q>} .<dt> <Dd>, <Qm>

Assembler Symbols

- <c>

For the "A1 Signed result" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 Signed result" variant: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the operand, encoded in “(op :: size)”:

op	size	<dt>
10	00	S16
10	01	S32
10	10	S64
1x	11	RESERVED
11	00	U16
11	01	U32
11	10	U64

- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of [VQMOVN, VQMOVUN](#) gives the operational pseudocode for this instruction.

VQRSHRN, VQRSHRUN

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the rounded results in a doubleword vector.

For truncated results, see [VQSHRN and VQSHRUN](#).

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	!= 000xxx						Vd				1	0	0	op	0	1	M	1	Vm			
imm6																								L		Q					

Encoding for the Signed result variant

Applies when (op == 1)

VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Encoding for the Unsigned result variant

Applies when (U == 1 && op == 0)

VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VRSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	!= 000xxx						Vd				1	0	0	op	0	1	M	1	Vm			
imm6																								L	Q						

Encoding for the Signed result variant

Applies when (op == 1)

VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Encoding for the Unsigned result variant

Applies when (U == 1 && op == 0)

VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VRSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For the "A1 Signed result" and "A1 Unsigned result" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1 Signed result" and "T1 Unsigned result" variants: see *Standard assembler syntax fields*.
<q> See *Standard assembler syntax fields*.
<type> For the "A1 Signed result" and "T1 Signed result" variants: is the data type for the elements of the vectors, encoded in "U":

U	<type>
0	S
1	U

For the "A1 Unsigned result" and "T1 Unsigned result" variants: is S when "U" is 1, and is encoded in "U".
<size> Is the data size for the elements of the vectors, encoded in "imm6":

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let opelt : bits(2*esize) = Qin(m>>1)[e*(2*esize)];
        let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
        let rshrn : integer = RShr(element, shift_amount, round);
        let (result, sat) : (bits(esize), boolean) = SatQ(esize)(rshrn, dest_unsigned);
        D(d)[e*esize] = result;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```

VQRSHRUN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned rounded results in a doubleword vector.

This is a pseudo-instruction of [VQMOVN, VQMOVUN](#). This means:

- The encodings in this description are named to match the encodings of [VQMOVN, VQMOVUN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VQMOVN, VQMOVUN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size			1	0	Vd				0	0	1	0	0		1	M	0	Vm		
opc1														op																		

Encoding for the Unsigned result variant

VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd				0	0	1	0	0		1	M	0	Vm		
opc1														op																		

Encoding for the Unsigned result variant

VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

- <c>

For the "A1 Unsigned result" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1 Unsigned result" variant: see [Standard assembler syntax fields](#).
- <q>

See [Standard assembler syntax fields](#).
- <dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of [VQMOVN, VQMOVUN](#) gives the operational pseudocode for this instruction.

VQSHL (register)

Vector Saturating Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results are truncated. For rounded results, see [VQRSHL](#).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn				Vd				0	1	0	0	N	Q	M	1	Vm				
												opc												o1							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	0	D	size	Vn					Vd					0	1	0	0	N	Q	M	1	Vm				
												opc												o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```


Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the vectors, encoded in “(U :: size)”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var sat : boolean;
            var result : bits(esize);
            if shift >= 0 then
                (result,sat) = SatQ{esize}(element << shift, unsigned);
            else
                (result,sat) = SatQ{esize}(element >> -shift, unsigned);
            end;
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```

VQSHL, VQSHLU (immediate)

Vector Saturating Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in a second vector.

The operand elements must all be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are the same size as the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR.QC*, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPT* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	1	1	op	L	Q	M	1	Vm			

Encoding for the 64-bit SIMD vector, signed result variant

Applies when (op == 1 && Q == 0 && !(imm6[5:3] == 000 && L == 0))

VQSHL{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 64-bit SIMD vector, unsigned result variant

Applies when (U == 1 && op == 0 && Q == 0 && !(imm6[5:3] == 000 && L == 0))

VQSHLU{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector, signed result variant

Applies when (op == 1 && Q == 1 && !(imm6[5:3] == 000 && L == 0))

VQSHL{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Encoding for the 128-bit SIMD vector, unsigned result variant

Applies when (U == 1 && op == 0 && Q == 1 && !(imm6[5:3] == 000 && L == 0))

VQSHLU{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	1	1	op	L	Q	M	1	Vm			

Encoding for the 64-bit SIMD vector, signed result variant

Applies when (op == 1 && Q == 0 && !(imm6[5:3] == 000 && L == 0))

VQSHL{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 64-bit SIMD vector, unsigned result variant

Applies when (U == 1 && op == 0 && Q == 0 && !(imm6[5:3] == 000 && L == 0))

VQSHLU{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector, signed result variant

Applies when (op == 1 && Q == 1 && !(imm6[5:3] == 000 && L == 0))

VQSHL{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Encoding for the 128-bit SIMD vector, unsigned result variant

Applies when (U == 1 && op == 0 && Q == 1 && !(imm6[5:3] == 000 && L == 0))

VQSHLU{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c> For the "A1 128-bit SIMD vector, signed result", "A1 128-bit SIMD vector, unsigned result", "A1 64-bit SIMD vector, signed result", and "A1 64-bit SIMD vector, unsigned result" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 128-bit SIMD vector, signed result", "T1 128-bit SIMD vector, unsigned result", "T1 64-bit SIMD vector, signed result", and "T1 64-bit SIMD vector, unsigned result" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<type> Is the data type for the elements of the vectors, encoded in "U":

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in "(L :: imm6)":

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
            let (result, sat) : (bits(esize), boolean) = SatQ(esize)(element << shift_amount,
                                                                    dest_unsigned);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```


VQSHRN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed truncated results in a doubleword vector.

This is a pseudo-instruction of [VQMOVN, VQMOVUN](#). This means:

- The encodings in this description are named to match the encodings of [VQMOVN, VQMOVUN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VQMOVN, VQMOVUN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd					0	0	1	0	1		x	M	0	Vm		
opc1														op																		

Encoding for the Signed result variant

VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd				0	0	1	0	1	x	M	0	Vm				
opc1														op																	

Encoding for the Signed result variant

VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

- <c>
- For the "A1 Signed result" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1 Signed result" variant: see [Standard assembler syntax fields](#).
- <q>
- See [Standard assembler syntax fields](#).
- <dt>
- Is the data type for the elements of the operand, encoded in “(op :: size)”:

op	size	<dt>
10	00	S16
10	01	S32
10	10	S64
1x	11	RESERVED
11	00	U16
11	01	U32
11	10	U64

- <Dd>
- Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qm>
- Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of [VQMOVN, VQMOVUN](#) gives the operational pseudocode for this instruction.

VQSHRN, VQSHRUN

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the truncated results in a doubleword vector.

For rounded results, see [VQRSHRN and VQRSHRUN](#).

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	!= 000xxx						Vd				1	0	0	op	0	0	M	1	Vm			
imm6																								L				Q			

Encoding for the Signed result variant

Applies when (op == 1)

VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Encoding for the Unsigned result variant

Applies when (U == 1 && op == 0)

VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	!= 000xxx						Vd				1	0	0	op	0	0	M	1	Vm			
imm6																								L				Q			

Encoding for the Signed result variant

Applies when (op == 1)

VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Encoding for the Unsigned result variant

Applies when (U == 1 && op == 0)

VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For the "A1 Signed result" and "A1 Unsigned result" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1 Signed result" and "T1 Unsigned result" variants: see *Standard assembler syntax fields*.
<q> See *Standard assembler syntax fields*.
<type> For the "A1 Signed result" and "T1 Signed result" variants: is the data type for the elements of the vectors, encoded in "U":

U	<type>
0	S
1	U

For the "A1 Unsigned result" and "T1 Unsigned result" variants: is S when "U" is 1, and is encoded in "U".
<size> Is the data size for the elements of the vectors, encoded in "imm6":

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let opelt : bits(2*esize) = Qin(m>>1)[e:(2*esize)];
        let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
        let (result, sat) : (bits(esize), boolean) = SatQ(esize)(element >> shift_amount,
                                                                    dest_unsigned);
        D(d)[e*:esize] = result;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```

VQSHRUN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned truncated results in a doubleword vector.

This is a pseudo-instruction of [VQMOVN, VQMOVUN](#). This means:

- The encodings in this description are named to match the encodings of [VQMOVN, VQMOVUN](#).
 - The assembler syntax is used only for assembly, and is not used on disassembly.
 - The description of [VQMOVN, VQMOVUN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.
- It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd					0	0	1	0	0		1	M	0	Vm		
opc1														op																		

Encoding for the Unsigned result variant

VQSHRUN{<c>} {<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

[VQMOVUN](#){<c>} {<q>}.<dt> <Dd>, <Qm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd				0	0	1	0	0	1	M	0	Vm				
opc1														op																	

Encoding for the Unsigned result variant

VQSHRUN{<c>} {<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

[VQMOVUN](#){<c>} {<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

<c>	For the "A1 Unsigned result" variant: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 Unsigned result" variant: see Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<dt>	Is the data type for the elements of the operand, encoded in “size”: <table><tr><th>size</th><th><dt></th></tr><tr><td>00</td><td>S16</td></tr><tr><td>01</td><td>S32</td></tr><tr><td>10</td><td>S64</td></tr><tr><td>11</td><td>RESERVED</td></tr></table>	size	<dt>	00	S16	01	S32	10	S64	11	RESERVED
size	<dt>										
00	S16										
01	S32										
10	S64										
11	RESERVED										
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.										
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.										

Operation

The description of [VQMOVN, VQMOVUN](#) gives the operational pseudocode for this instruction.

VQSUB

Vector Saturating Subtract subtracts the elements of the second operand vector from the corresponding elements of the first operand vector, and places the results in the destination vector. Signed and unsigned operations are distinct. The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size			Vn			Vd			0			0	1	0	N	Q	M	1	Vm		
opc																					o1										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size			Vn			Vd			0	0	1	0	N	Q	M	1	Vm				
opc																					o1										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).

<q>

See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the vectors, encoded in “(U :: size)”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

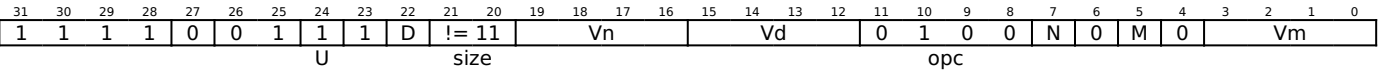
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let diff : integer = element1 - element2;
            let (res, sat) : (bits(esize), boolean) = SatQ{esize}(diff, unsigned);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;

```

VRADDHN

Vector Rounding Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are rounded. For truncated results, see [VADDHN](#).
The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.
Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



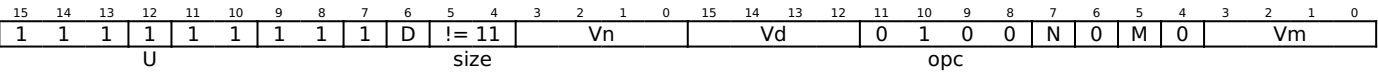
Encoding

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1



Encoding

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

- <c> For the "A1" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1" variant: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let result : integer = (RShr(UInt(Qin(n>>1)[e:(2*esize)] +
            Qin(m>>1)[e:(2*esize)]), esize, round));
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRECPE

Vector Reciprocal Estimate finds an approximate reciprocal of each element in the operand vector, and places the results in the destination vector. The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see [Floating-point reciprocal square root estimate and step](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd			0		1	0	F	0	Q	M	0	Vm			
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0')) || size IN {'00', '11'} then
    Undefined();
end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	1	Vd				0	1	0	F	0	Q	M	0	Vm			
opc1																																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0')) || size IN {'00', '11'} then
    Undefined();
end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- <dt>

Is the data type for the elements of the vectors, encoded in “(F :: size)”:

F	size	<dt>
0	10	U32
1	01	F16
1	10	F32
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see [Floating-point reciprocal estimate and step](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if floating_point then
                D(d+r)[e*:esize] = FPRecipEstimate{esize}(D(m+r)[e*:esize], fpcr);
            else
                D(d+r)[e*:esize] = UnsignedRecipEstimate{esize}(D(m+r)[e*:esize]);
            end;
        end;
    end;
end;
```

VRECPS

Vector Reciprocal Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 2.0, and places the results into the elements of the destination vector.

The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			
U												opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			
U												opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- <dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see [Floating-point reciprocal estimate and step](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = FPREcipStep{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize]);
        end;
    end;
end;
```

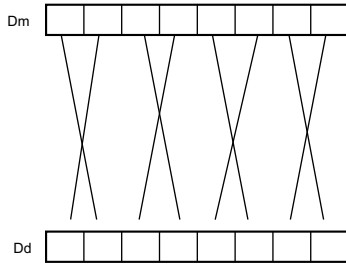

VREV16

Vector Reverse in halfwords reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows an example of the operation of VREV16 doubleword operation.

VREV16.8, doubleword



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	0	Vd			0			0	0	1	0	Q	M	0	Vm			
												opc1																op			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```

if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0		Vd				0	0	0	1	0	Q	M	0		Vm		
opc1															op																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```

if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the operand, encoded in “size”:
- | size | <dt> |
|------|----------|
| 00 | 8 |
| 01 | RESERVED |
| 1x | RESERVED |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var result : bits(64);
    var element : integer;
    var rev_element : integer;
    for r = 0 to regs-1 do
        element = 0;
        for c = 0 to containers-1 do
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1 do
                result[rev_element*esize] = D(m+r)[element*esize];
                element = element + 1;
                rev_element = rev_element - 1;
            end;
            end;
        D(d+r) = result;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VREV32

Vector Reverse in words reverses the order of 8-bit or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows an example of the operation of VREV32 doubleword operations.

VREV32.8, doubleword

VREV32.16, doubleword

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0			0	0	0		1	Q	M	0	Vm	
opc1														op																		

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0		Vd				0	0	0	0	1	Q	M	0	Vm			
opc1														op																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```

if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the operand, encoded in “size”:
- | size | <dt> |
|------|----------|
| 00 | 8 |
| 01 | 16 |
| 1x | RESERVED |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var result : bits(64);
    var element : integer;
    var rev_element : integer;
    for r = 0 to regs-1 do
        element = 0;
        for c = 0 to containers-1 do
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1 do
                result[rev_element*esize] = D(m+r)[element*esize];
                element = element + 1;
                rev_element = rev_element - 1;
            end;
        end;
        D(d+r) = result;
    end;
end;

```

Operational information

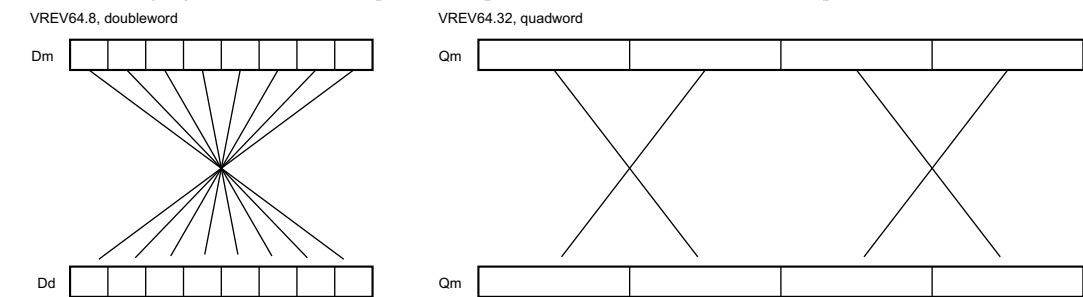
This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VREV64

Vector Reverse in doublewords reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows an example of the operation of VREV64 doubleword operations.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0			0	0	0		0	Q	M	0	Vm		
opc1														op																			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	0	0	Vd				0	0	0	0	0	Q	M	0	Vm				
opc1														op																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```

if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the operand, encoded in “size”:
- | size | <dt> |
|------|----------|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | RESERVED |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var result : bits(64);
    var element : integer;
    var rev_element : integer;
    for r = 0 to regs-1 do
        element = 0;
        for c = 0 to containers-1 do
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1 do
                result[rev_element*esize] = D(m+r)[element*esize];
                element = element + 1;
                rev_element = rev_element - 1;
            end;
        end;
        D(d+r) = result;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VRHADD

Vector Rounding Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The results of the halving operations are rounded. For truncated results, see [VHADD](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size						Vn				Vd		0	0	0	1	N	Q	M	0		Vm
												opc																o1			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size						Vn				Vd		0	0	0	1	N	Q	M	0		Vm
												opc																o1			

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

<q>
<dt>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
See [Standard assembler syntax fields](#).

Is the data type for the elements of the operands, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(usize) = D(n+r)[e*:usize];
            let op2elt : bits(usize) = D(m+r)[e*:usize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (element1 + element2 + 1) >> 1;
            D(d+r)[e*:usize] = result[usize-1:0];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VRINTA (Advanced SIMD)

Vector Round floating-point to integer towards Nearest with Ties to Away rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	1	D	1	1	size			1	0	Vd					0	1	0	1	0	Q	M	0	Vm			
opc1												op																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTA{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTA{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd					0	1	0	1	0	Q	M	0	Vm			
opc1															op																		

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTA{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTA{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock](#)(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD two registers misc](#) for the T32 instruction set, or [Advanced SIMD two registers misc](#) for the A32 instruction set.

Assembler Symbols

- <q>

See [Standard assembler syntax fields](#).
- <dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  for e = 0 to elements-1 do
    let operand1 : bits(esize) = D(m+r)[e*:esize];
    let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
    D(d+r)[e*:esize] = result;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTA (floating-point)

Round floating-point to integer to Nearest with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	0	0	Vd				1	0	size		0	1	M	0	Vm			
op1													RM			op															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VRINTA{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTA{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTA{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	0	0	0	Vd				1	0	size	0	1	M	0	Vm			
op1													RM			op															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VRINTA{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTA{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTA{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 16 =>
    H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
  when 32 =>
    S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
  when 64 =>
    D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTM (Advanced SIMD)

Vector Round floating-point to integer towards -Infinity rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	1	D	1	1	size			1	0	Vd					0	1	1	0	1	Q	M	0	Vm			
opc1												op																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTM{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTM{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd					0	1	1	0	1	Q	M	0	Vm		
opc1															op																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTM{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTM{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock\(\)](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD two registers misc](#) for the T32 instruction set, or [Advanced SIMD two registers misc](#) for the A32 instruction set.

Assembler Symbols

- <q>

See [Standard assembler syntax fields](#).
- <dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  for e = 0 to elements-1 do
    let operand1 : bits(esize) = D(m+r)[e*:esize];
    let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
    D(d+r)[e*:esize] = result;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTM (floating-point)

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	1	1	Vd				1	0	size	0	1	M	0	Vm				
op1													RM				op														

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VRINTM{<q>}.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTM{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTM{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	1	1	Vd				1	0	size	0	1	M	0	Vm				
op1															RM		op														

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VRINTM{<q>}.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTM{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTM{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock](#)(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 16 =>
    H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
  when 32 =>
    S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
  when 64 =>
    D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTN (Advanced SIMD)

Vector Round floating-point to integer to Nearest rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																								
1	1	1	1	0	0	1	1	1	D	1	1	size		1		0		Vd				0		1		0		0		0		Q	M	0	Vm																				
												opc1																												op															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTN{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTN{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd					0	1	0	0	0	Q	M	0	Vm			
opc1															op																		

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTN{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTN{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock\(\)](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD two registers misc](#) for the T32 instruction set, or [Advanced SIMD two registers misc](#) for the A32 instruction set.

Assembler Symbols

- <q>

See [Standard assembler syntax fields](#).
- <dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  for e = 0 to elements-1 do
    let operand1 : bits(esize) = D(m+r)[e*:esize];
    let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
    D(d+r)[e*:esize] = result;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	0	1	Vd				1	0	size	0	1	M	0	Vm				
op1													RM			op															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VRINTN{<q>}.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTN{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTN{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	0	1	D	1	1	1	0	0	1	Vd				1	0	size	0	1	M	0	Vm					
op1															RM			op														

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VRINTN{<q>}.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTN{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTN{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock](#)(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 16 =>
    H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
  when 32 =>
    S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
  when 64 =>
    D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTP (Advanced SIMD)

Vector Round floating-point to integer towards +Infinity rounds a vector of floating-point values to integral floating-point values of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size			1	0	Vd				0	1	1	1	1	Q	M	0	Vm			
opc1												op																				

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTP{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTP{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd				0	1	1	1	1	Q	M	0	Vm			
opc1															op																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTP{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTP{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock\(\)](#), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD two registers misc](#) for the T32 instruction set, or [Advanced SIMD two registers misc](#) for the A32 instruction set.

Assembler Symbols

- <q>

See [Standard assembler syntax fields](#).
- <dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  for e = 0 to elements-1 do
    let operand1 : bits(esize) = D(m+r)[e*:esize];
    let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
    D(d+r)[e*:esize] = result;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	1	0	Vd				1	0	size	0	1	M	0	Vm				
op1										RM				op																	

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VRINTP{<q>}.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTP{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTP{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	1	0	Vd				1	0	size	0	1	M	0	Vm				
op1															RM		op														

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VRINTP{<q>}.F16 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTP{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTP{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If [InITBlock](#)(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 16 =>
    H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
  when 32 =>
    S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
  when 64 =>
    D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTR

Round floating-point to integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in the FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	1	1	0	Vd				1	0	size	0	1	M	0	Vm				
cond									o1				opc2				op														

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	0	Vd				1	0	size	0	1	M	0	Vm				
									o1				opc2				op														

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding\_ZERO else FPRoundingMode(fpcr);
    case esize of
        when 16 =>
            H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
        when 32 =>
            S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
        when 64 =>
            D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTX (Advanced SIMD)

Vector round floating-point to integer inexact rounds a vector of floating-point values to integral floating-point values of the same size, using the Round to Nearest rounding mode, and raises the Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd					0	1	0	0	1	Q	M	0	Vm				
opc1																opc2																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTX{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTX{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
let rounding : FPRounding = FPRounding\_TIEEVEN;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd				0	1	0	0	1	Q	M	0	Vm			
opc1																opc2																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRINTX{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTX{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
let rounding : FPRounding = FPRounding\_TIEEVEN;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the vectors, encoded in "size": <table><tr><th>size</th><th><dt></th></tr><tr><td>01</td><td>F16</td></tr><tr><td>10</td><td>F32</td></tr></table>	size	<dt>	01	F16	10	F32
size	<dt>						
01	F16						
10	F32						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  for e = 0 to elements-1 do
    let operand1 : bits(esize) = D(m+r)[e*:esize];
    let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
    D(d+r)[e*:esize] = result;
  end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTX (floating-point)

Round floating-point to integer inexact rounds a floating-point value to an integral floating-point value of the same size, using the rounding mode specified in the FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				1	1	1	0	1	D	1	1	0	1	1	1	Vd				1		0	size		0	1	M	0	Vm			
cond								o1				opc2								o3												

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	size	0	1	M	0	Vm								
o1																opc2				o3															

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = FPRoundingMode(fpcr);
    case esize of
        when 16 =>
            H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
        when 32 =>
            S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
        when 64 =>
            D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
    end;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTZ (Advanced SIMD)

Vector round floating-point to integer towards Zero rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd				0	1	0	1	1	Q	M	0	Vm				
opc1																opc2															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VRINTZ{<q>}.<dt> <Dd>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

`VRINTZ{<q>}.<dt> <Qd>, <Qm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
let rounding : FPRounding = FPRounding\_ZERO;
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd				0	1	0			1	1	Q	M	0	Vm		
opc1																opc2																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VRINTZ{<q>}.<dt> <Dd>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

`VRINTZ{<q>}.<dt> <Qd>, <Qm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then Undefined(); end;
let rounding : FPRounding = FPRounding\_ZERO;
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the vectors, encoded in "size": <table><tr><th>size</th><th><dt></th></tr><tr><td>01</td><td>F16</td></tr><tr><td>10</td><td>F32</td></tr></table>	size	<dt>	01	F16	10	F32
size	<dt>						
01	F16						
10	F32						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  for e = 0 to elements-1 do
    let operand1 : bits(esize) = D(m+r)[e*:esize];
    let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);
    D(d+r)[e*:esize] = result;
  end;
end;
```


VRINTZ (floating-point)

Round floating-point to integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	1	1	0	Vd				1	0	size		1	1	M	0	Vm			
cond									o1				opc2				op														

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	0	Vd				1	0	size	1	1	M	0	Vm				
									o1				opc2				op														

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding\_ZERO else FPRoundingMode(fpcr);
    case esize of
        when 16 =>
            H(d) = FPRoundInt{16}(H(m), fpcr, rounding, exact);
        when 32 =>
            S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
        when 64 =>
            D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see `VSHL`.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the `CPACR`, `NSACR`, and `HCPTR` registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	U	0	D	size			Vn			Vd			0			1	0	1	N	Q	M	0	Vm				
opc																				o1													

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VRSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

`VRSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size			Vn			Vd			0	1	0	1	N	Q	M	0	Vm				
opc																				o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VRSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

`VRSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the vectors, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var result : integer;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            if shift >= 0 then // left shift
                result = element << shift;
            else // rounding right shift
                shift = -shift;
                result = (element + (1 << (shift - 1))) >> shift;
            end;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VRSHR

Vector Rounding Shift Right takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see [VSHR](#).

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	1	0	L	Q	M	1	Vm			
																opc															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VRSHR{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VRSHR{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	1	0	L	Q	M	1	Vm			
																opc															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VRSHR{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VRSHR{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

<q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

<type>

See *Standard assembler syntax fields*.

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “(L :: imm6)”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm>

Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = RShr(element, shift_amount, round);
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VRSHR (zero)

Vector Rounding Shift Right copies the contents of one SIMD register to another.

This is a pseudo-instruction of [VORR \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VORR \(register\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VORR \(register\)](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	0	Vn			Vd			0			0	1	N	Q	M	1	Vm				
U							size				opc										o1										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	0	Vn			Vd			0			0	0	1	N	Q	M	1	Vm			
U									size				opc										o1								

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<dt>	Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

Operation

The description of [VORR \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VORR \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHRN

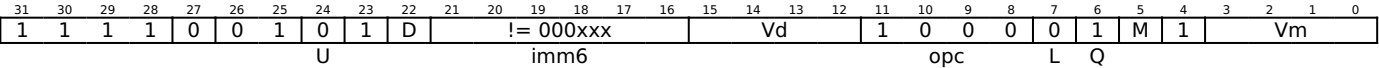
Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see [VSHRN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



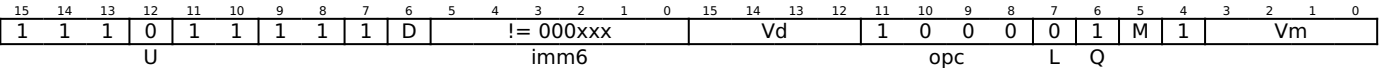
Encoding

VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1



Encoding

VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1" variant: see [Standard assembler syntax fields](#).
- <size>

See [Standard assembler syntax fields](#).
- Is the data size for the elements of the vectors, encoded in “imm6”:
- | imm6 | <size> |
|--------|--------|
| 001xxx | 16 |
| 01xxxx | 32 |
| 1xxxxx | 64 |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- <imm>

Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let result : integer = RShr(UInt(Qin(m>>1)[e:(2*esize)]), shift_amount, round);
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHRN (zero)

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector.

This is a pseudo-instruction of [VMOVN](#). This means:

- The encodings in this description are named to match the encodings of [VMOVN](#).
 - The assembler syntax is used only for assembly, and is not used on disassembly.
 - The description of [VMOVN](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.
- It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	0	0	0	M	0	Vm			
opc1														opc2						Q												

Encoding for the A1 variant

VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

[VMOVN](#){<c>}{<q>}.<dt> <Dd>, <Qm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd				0	0	1	0	0	0	M	0	Vm				
opc1														opc2						Q											

Encoding for the T1 variant

VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

[VMOVN](#){<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

- <c>
- For the "A1" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1" variant: see [Standard assembler syntax fields](#).
- <q>
- See [Standard assembler syntax fields](#).
- <dt>
- Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

- <Dd>
- Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qm>
- Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of [VMOVN](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VMOVN](#) gives the operational information for this instruction.

VRSQRTE

Vector Reciprocal Square Root Estimate finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	1		Vd				0	1	0	F	1	Q	M	0		Vm		
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if (size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0')) || size IN {'00', '11'} then
    Undefined();
end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	1		Vd				0	1	0	F	1	Q	M	0		Vm		
opc1																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0') then Undefined(); end;
if size IN {'00', '11'} then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the vectors, encoded in “(F :: size)”:

F	size	<dt>
0	10	U32
1	01	F16
1	10	F32

- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see [Floating-point reciprocal estimate and step](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if floating_point then
                D(d+r)[e*:esize] = FPRSqrtEstimate{esize}(D(m+r)[e*:esize], fpcr);
            else
                D(d+r)[e*:esize] = UnsignedRSqrtEstimate{esize}(D(m+r)[e*:esize]);
            end;
        end;
    end;
end;
```

VRSQRTS

Vector Reciprocal Square Root Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the elements of the destination vector.
The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	0	1	0	0	D	1	sz	Vn				Vd				1				1	1	1	N	Q	M	1	Vm			
U												opc								o1														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			
U												opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.						
	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see [Floating-point reciprocal estimate and step](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = FPRSqrtStep{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize]);
        end;
    end;
end;
```

VRSRA

Vector Rounding Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the rounded results into the destination vector. For truncated results, see [VSRRA](#).

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	1	1	L	Q	M	1	Vm			
																opc															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VRSRA{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VRSRA{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	1	1	L	Q	M	1	Vm			
opc																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VRSRA{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VRSRA{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

<q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

<type>

See *Standard assembler syntax fields*.

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “(L :: imm6)”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm>

Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = RShr(element, shift_amount, round);
            D(d+r)[e*:esize] = D(d+r)[e*:esize] + result;
        end;
    end;
end;
```

Operational information

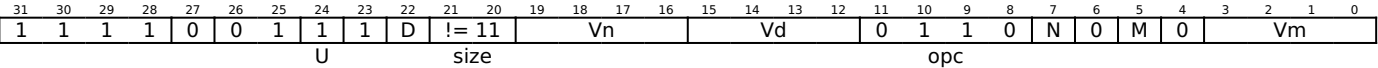
This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VRSUBHN

Vector Rounding Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are rounded. For truncated results, see [VSUBHN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#). It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



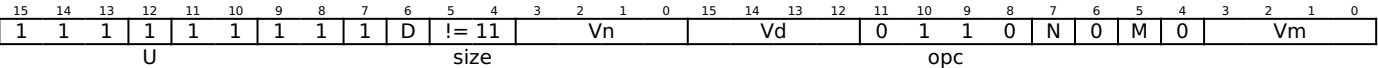
Encoding

VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1



Encoding

VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1" variant: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the operands, encoded in “size”:
- | size | <dt> |
|------|------|
| 00 | I16 |
| 01 | I32 |
| 10 | I64 |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let result : integer = (RShr(UInt(Qin(n>>1)[e:(2*esize)] -
            Qin(m>>1)[e:(2*esize)]), esize, round));
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSDOT (by element)

Dot Product index form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	1	0	Vn			Vd			1			1	0	1	N	Q	M	0	Vm			
op1									op2			op4									U										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let signed : boolean = (U=='0');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm[3:0]);
let index : integer = UInt(M);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

T1
(FEAT_DotProd)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	1	0	Vn			Vd			1	1	0	1	N	Q	M	0	Vm					
op1									op2			op4									U										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let signed : boolean = (U=='0');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm[3:0]);
let index : integer = UInt(M);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```

var operand1 : bits(64);
let operand2 : bits(64) = D(m);
var result : bits(64);
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    operand1 = D(n+r);
    result = D(d+r);
    var element1, element2 : integer;
    for e = 0 to 1 do
        var res : integer = 0;
        for i = 0 to 3 do
            if signed then
                element1 = SInt(operand1[(4 * e + i):(esize DIV 4)]);
                element2 = SInt(operand2[(4 * index + i):(esize DIV 4)]);
            else
                element1 = UInt(operand1[(4 * e + i):(esize DIV 4)]);
                element2 = UInt(operand2[(4 * index + i):(esize DIV 4)]);
            end;
            res = res + element1 * element2;
        end;
        result[e*esize] = result[e*esize] + res;
    end;
    D(d+r) = result;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSDOT (vector)

Dot Product vector form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an **OPTIONAL** instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	0	Vn			Vd			1	1	0	1	N	Q	M	0	Vm					
op1						op2						op3				op4				U											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let signed : boolean = U=='0';
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_DotProd)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	0	Vn			Vd			1	1	0	1	N	Q	M	0	Vm					
op1						op2						op3				op4				U											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if !InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let signed : boolean = U=='0';
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>

See *Standard assembler syntax fields*.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    operand1 = D(n+r);
    operand2 = D(m+r);
    result = D(d+r);
    var element1, element2 : integer;
    for e = 0 to 1 do
        var res : integer = 0;
        for i = 0 to 3 do
            if signed then
                element1 = SInt(operand1[(4 * e + i):(esize DIV 4)]);
                element2 = SInt(operand2[(4 * e + i):(esize DIV 4)]);
            else
                element1 = UInt(operand1[(4 * e + i):(esize DIV 4)]);
                element2 = UInt(operand2[(4 * e + i):(esize DIV 4)]);
            end;
            res = res + element1 * element2;
        end;
        result[e*esize] = result[e*esize] + res;
    end;
    D(d+r) = result;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSELEQ, VSELGE, VSELGT, VSELVS

Floating-point conditional select allows the destination register to take the value in either one or the other source register according to the condition codes in the [APSR](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	cc	Vn			Vd			1	0	size	N	0	M	0	Vm							

Encoding for the Equal, half-precision scalar variant (FEAT_FP16)

Applies when (cc == 00 && size == 01)

VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Equal, single-precision scalar variant

Applies when (cc == 00 && size == 10)

VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Equal, double-precision scalar variant

Applies when (cc == 00 && size == 11)

VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Encoding for the Greater than or Equal, half-precision scalar variant (FEAT_FP16)

Applies when (cc == 10 && size == 01)

VSELGE.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Greater than or Equal, single-precision scalar variant

Applies when (cc == 10 && size == 10)

VSELGE.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Greater than or Equal, double-precision scalar variant

Applies when (cc == 10 && size == 11)

VSELGE.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Encoding for the Greater than, half-precision scalar variant (FEAT_FP16)

Applies when (cc == 11 && size == 01)

VSELGT.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Greater than, single-precision scalar variant

Applies when (cc == 11 && size == 10)

VSELGT.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Greater than, double-precision scalar variant

Applies when (cc == 11 && size == 11)

VSELGT.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Encoding for the Unordered, half-precision scalar variant (FEAT_FP16)

Applies when (cc == 01 && size == 01)

VSELVS.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Unordered, single-precision scalar variant

Applies when (cc == 01 && size == 10)

VSELVS.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Encoding for the Unordered, double-precision scalar variant

Applies when (cc == 01 && size == 11)

VSELVS.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let condition : bits(4) = cc::(cc[1] XOR cc[0])::'0';
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	cc	Vn				Vd				1	0	size	N	0	M	0	Vm					

Encoding for the Equal, half-precision scalar variant (FEAT_FP16)

Applies when (cc == 00 && size == 01)

VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Encoding for the Equal, single-precision scalar variant

Applies when (cc == 00 && size == 10)

VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Encoding for the Equal, double-precision scalar variant

Applies when (cc == 00 && size == 11)

VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

Encoding for the Greater than or Equal, half-precision scalar variant (FEAT_FP16)

Applies when (cc == 10 && size == 01)

VSELGE.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Encoding for the Greater than or Equal, single-precision scalar variant

Applies when (cc == 10 && size == 10)

VSELGE.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Encoding for the Greater than or Equal, double-precision scalar variant

Applies when (cc == 10 && size == 11)

VSELGE.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

Encoding for the Greater than, half-precision scalar variant (FEAT_FP16)

Applies when (cc == 11 && size == 01)

VSELGT.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Encoding for the Greater than, single-precision scalar variant

Applies when (cc == 11 && size == 10)

VSELGT.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Encoding for the Greater than, double-precision scalar variant

Applies when (cc == 11 && size == 11)

VSELGT.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

Encoding for the Unordered, half-precision scalar variant (FEAT_FP16)

Applies when (cc == 01 && size == 01)

```
VSELVS.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Unordered, single-precision scalar variant

Applies when (cc == 01 && size == 10)

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Unordered, double-precision scalar variant

Applies when (cc == 01 && size == 11)

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let condition : bits(4) = cc::(cc[1] XOR cc[0])::'0';
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
case esize of
  when 16 =>
    H(d) = if ConditionHolds(condition) then H(n) else H(m);
  when 32 =>
    S(d) = if ConditionHolds(condition) then S(n) else S(m);
  when 64 =>
    D(d) = if ConditionHolds(condition) then D(n) else D(m);
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHL (immediate)

Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			
U										opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSHL{<c>}{<q>}.I<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSHL{<c>}{<q>}.I<size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			
U										opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSHL{<c>}{<q>}.I<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSHL{<c>}{<q>}.I<size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).

<q>

See *Standard assembler syntax fields*.

<size>

Is the data size for the elements of the vectors, encoded in "(L :: imm6)":

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm>

Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = LSL{esize}(D(m+r)[e*:esize], shift_amount);
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VSHL (register)

Vector Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

For a rounding shift, see [VRSHL](#).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	U	0	D	size				Vn				Vd				0	1	0	0	N	Q	M	0	Vm			
opc																										o1							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	U	1	1	1	1	0	D	size				Vn				Vd				0	1	0	0	N	Q	M	0	Vm			
opc																										o1							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see [Standard assembler syntax fields](#).

See [Standard assembler syntax fields](#).

Is the data type for the elements of the vectors, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var result : integer;
            if shift >= 0 then
                result = element << shift;
            else
                result = element >> -shift;
            end;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHLL

Vector Shift Left Long takes each element in a doubleword vector, left shifts them by an immediate value, and places the results in a quadword vector.

The operand elements can be:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit untyped integers, maximum shift only.

The result elements are twice the length of the operand elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				1	0	1	0	0	0	M	1	Vm			
											opc						L				Q										

Encoding

Applies when (imm6 IN {100001, 10001x, 1001xx, 101xxx, 11xxxx, 010001, 01001x, 0101xx, 011xxx, 001001, 00101x, 0011xx})

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if imm6 IN {'001000', '010000', '100000'} then See("VMOVL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(imm6) - esize;
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	1	1	0	0	M	0	Vm			
													opc1				opc2				Q										

Encoding

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

Decode for this encoding

```
if size == '11' || Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let shift_amount : integer = esize;
let unsigned : boolean = FALSE; // Or TRUE without change of functionality
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				1	0	1	0	0	0	M	1	Vm			
											opc						L				Q										

Encoding

Applies when (imm6 IN {100001, 10001x, 1001xx, 101xxx, 11xxxx, 010001, 01001x, 0101xx, 011xxx, 001001, 00101x, 0011xx})

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if imm6 IN {'001000', '010000', '100000'} then See("VMOVL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(imm6) - esize;
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0		Vd		0	0	1	1	0	0	M	0		Vm				
opc1																opc2								Q							

Encoding

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

Decode for this encoding

```
if size == '11' || Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let shift_amount : integer = esize;
let unsigned : boolean = FALSE; // Or TRUE without change of functionality
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" and "A2" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" and "T2" variants: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<size>

The data size for the elements of the operand. The following table shows the permitted values and their encodings:

<size>	Encoding T1/A1	Encoding T2/A2
8	Encoded as imm6<5:3> =0b001	Encoded as size =0b00
16	Encoded as imm6<5:4> =0b01	Encoded as size =0b01
32	Encoded as imm6<5> = 1	Encoded as size =0b10

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm>

The immediate value. <imm> must lie in the range 1 to <size>, and:

- If <size> == <imm>, the encoding is T2/A2.
- Otherwise, the encoding is T1/A1, and:
 - If <size> == 8, <imm> is encoded in imm6<2:0>.
 - If <size> == 16, <imm> is encoded in imm6<3:0>.
 - If <size> == 32, <imm> is encoded in imm6<4:0>.

<type>

The data type for the elements of the operand. It must be one of:

S	Signed. In encoding T1/A1, encoded as U = 0.
U	Unsigned. In encoding T1/A1, encoded as U = 1.
I	Untyped integer, Available only in encoding T2/A2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let opelt : bits(esize) = Din(m)[e*:esize];
        let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
        let result : integer = element << shift_amount;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHR

Vector Shift Right takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see [VRSHR](#).

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	0	0	L	Q	M	1	Vm			
																														opc	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSHR{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSHR{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	0	0	L	Q	M	1	Vm			
																														opc	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSHR{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSHR{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “(L :: imm6)”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm>

Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = element >> shift_amount;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VSHR (zero)

Vector Shift Right copies the contents of one SIMD register to another.

This is a pseudo-instruction of [VORR \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VORR \(register\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VORR \(register\)](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	0	0	D	1	0	Vn				Vd				0				0	1	N	Q	M	1	Vm			
U							size				opc										o1												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	0	1	1	1	1	0	D	1	0	Vn				Vd				0				0	0	1	N	Q	M	1	Vm			
U									size				opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .
<dt>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

Operation

The description of [VORR \(register\)](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VORR \(register\)](#) gives the operational information for this instruction.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHRN

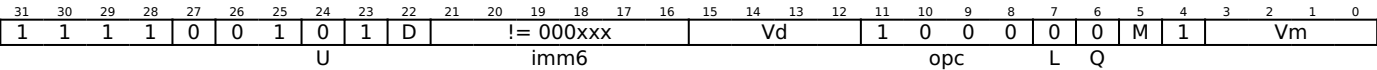
Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see [VRSHRN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



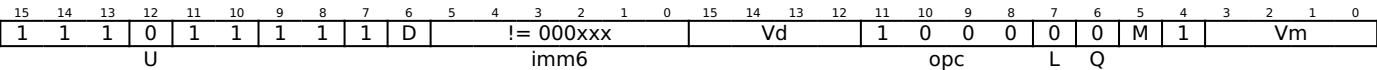
Encoding

VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1



Encoding

VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1" variant: see [Standard assembler syntax fields](#).
- <size>

See [Standard assembler syntax fields](#).
- Is the data size for the elements of the vectors, encoded in "imm6":
- | imm6 | <size> |
|--------|--------|
| 001xxx | 16 |
| 01xxxx | 32 |
| 1xxxxx | 64 |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- <imm>

Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let result : bits(2*esize) = LSR(Qin(m>>1)[e*:(2*esize)], shift_amount);
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHRN (zero)

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector.

This is a pseudo-instruction of [VMOVN](#). This means:

- The encodings in this description are named to match the encodings of [VMOVN](#).
 - The assembler syntax is used only for assembly, and is not used on disassembly.
 - The description of [VMOVN](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.
- It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	1	D	1	1	size			1	0	Vd					0	0	1	0	0	0	M	0	Vm			
opc1														opc2							Q												

Encoding for the A1 variant

VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd				0	0	1	0	0	0	M	0	Vm				
opc1														opc2							Q										

Encoding for the T1 variant

VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

- <c>
- For the "A1" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1" variant: see [Standard assembler syntax fields](#).
- <q>
- See [Standard assembler syntax fields](#).
- <dt>
- Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

- <Dd>
- Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qm>
- Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of [VMOVN](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VMOVN](#) gives the operational information for this instruction.

VSLI

Vector Shift Left and Insert takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	1	D	imm6						Vd						0	1	0	1	L	Q	M	1	Vm			
U										opc																							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSLI{<c>}{<q>}.<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSLI{<c>}{<q>}.<size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			
U										opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSLI{<c>}{<q>}.<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSLI{<c>}{<q>}.<size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

- <c>
- For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>
- For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<size>

Is the data size for the elements of the vectors, encoded in “(L :: imm6)”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm>

Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let mask : bits(esize) = LSL{}(Ones{esize}, shift_amount);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shifted_op : bits(esize) = LSL{}(D(m+r)[e*:esize], shift_amount);
            D(d+r)[e*:esize] = (D(d+r)[e*:esize] AND NOT(mask)) OR shifted_op;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

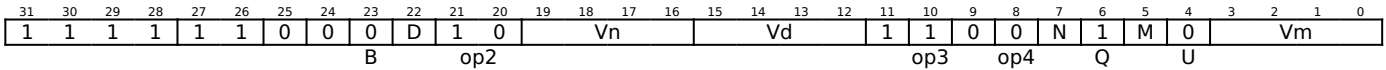
VSMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of signed 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32I8MM)



Encoding

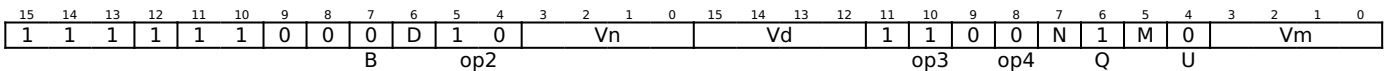
VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
  when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AA32I8MM)



Encoding

VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
  when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Qd>	Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();  
let operand1 : bits(128) = Q(n>>1);  
let operand2 : bits(128) = Q(m>>1);  
let addend : bits(128)   = Q(d>>1);  
  
Q(d>>1) = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSQRT

Square Root calculates the square root of the value in a floating-point register and writes the result to another floating-point register. Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPFXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	size	1	1	M	0	Vm				
cond								o1				opc2								o3											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	size	1	1	M	0	Vm				
o1																opc2				o3											

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>

Encoding for the Single-precision scalar variant

Applies when (size == 10)

VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 16 => H(d) = FPSqrt{16}(H(m), fpcr);
        when 32 => S(d) = FPSqrt{32}(S(m), fpcr);
        when 64 => D(d) = FPSqrt{64}(D(m), fpcr);
    end;
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSRA

Vector Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the truncated results into the destination vector. For rounded results, see [VRSRA](#).

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	0	1	L	Q	M	1	Vm			
opc																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSRA{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSRA{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	0	1	L	Q	M	1	Vm			
opc																															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSRA{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSRA{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “(L :: imm6)”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm>

Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = element >> shift_amount;
            D(d+r)[e*:esize] = D(d+r)[e*:esize] + result;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VSRI

Vector Shift Right and Insert takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	imm6						Vd				0	1	0	0	L	Q	M	1	Vm			
U										opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSRI{<c>}{<q>}.<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSRI{<c>}{<q>}.<size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	imm6						Vd				0	1	0	0	L	Q	M	1	Vm			
U										opc																					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSRI{<c>}{<q>}.<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSRI{<c>}{<q>}.<size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

- <c>
- For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>
- For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. See *Standard assembler syntax fields*.

<size>	Is the data size for the elements of the vectors, encoded in “(L :: imm6)”:																
	<table> <tr> <th>L</th><th>imm6</th><th><size></th></tr> <tr> <td>0</td><td>001xxx</td><td>8</td></tr> <tr> <td>0</td><td>01xxxx</td><td>16</td></tr> <tr> <td>0</td><td>1xxxxx</td><td>32</td></tr> <tr> <td>1</td><td>xxxxxx</td><td>64</td></tr> </table>	L	imm6	<size>	0	001xxx	8	0	01xxxx	16	0	1xxxxx	32	1	xxxxxx	64	
L	imm6	<size>															
0	001xxx	8															
0	01xxxx	16															
0	1xxxxx	32															
1	xxxxxx	64															
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.																
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.																
<imm>	Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.																
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.																
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.																

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let mask : bits(esize) = LSR(Ones(esize), shift_amount);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shifted_op : bits(esize) = LSR(D(m+r)[e*:esize], shift_amount);
            D(d+r)[e*:esize] = (D(d+r)[e*:esize] AND NOT(mask)) OR shifted_op;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VST1 (multiple single elements)

Store multiple single elements from one, two, three, or four registers stores elements to memory from one, two, three, or four registers, without interleaving. Every element of each register is stored. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) , [A3](#) and [A4](#)) and T32 ([T1](#) , [T2](#) , [T3](#) and [T4](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				0	1	1	1	size		align		Rm			
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				1	0	1	0	size		align		Rm			
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	Rn				Vd				0	1	1	0	size	align	Rm						
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn			Vd			0	0	1	0	size	align	Rm							
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn			Vd			0	1	1	1	size		align		Rm					
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn			Vd			1	0	1	0	size		align		Rm					
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				0	1	1	0	size		align		Rm			
										L		itype																			

Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

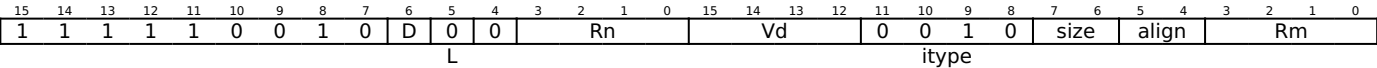
```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T4



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST1 \(multiple single elements\)](#).
Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c> For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", "A3 Post-indexed", "A4 Offset", and "A4 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", "T3 Post-indexed", "T4 Offset", and "T4 Post-indexed" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<size> Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32
11	64

<list> Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

- { <Dd> }
Single register. Selects the A1 and T1 encodings of the instruction.
- { <Dd>, <Dd+1> }
Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
- { <Dd>, <Dd+1>, <Dd+2> }
Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
- { <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }
Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64	64-bit alignment, encoded in the "align" field as 0b01.
128	128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.
256	256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode*.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if ebytes != 8 then
                MemU{8*ebytes}(address) = D(d+r)[e*:(8*ebytes)];
            else
                if !IsAlignedSize(address, ebytes) && AlignmentEnforced() then
                    let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
                    AArch32_Abort(fault);
                end;

                let data : bits(64) = D(d+r)[e*:64];
                if BigEndian(AccessType_ASIMD) then
                    MemU{32}(address) = data[63:32];
                    MemU{32}(address+4) = data[31:0];
                else
                    MemU{32}(address) = data[31:0];
                    MemU{32}(address+4) = data[63:32];
                end;
            end;

            address = address + ebytes;
        end;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 8*regs;
        end;
    end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST1 (single element from one lane)

Store single element from one lane of one register stores one element to memory from one element of a register. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) and [A3](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				0	0	0	0	index_align				Rm			
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				0	1	0	0	index_align				Rm						
L											size																N							

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				1 0 0 0				index_align				Rm								
L											size N																									

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn			Vd			0	0	0	0	index_align			Rm						
L										size N																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn			Vd			0	1	0	0	index_align				Rm					
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn			Vd			1	0	0	0	index_align				Rm					
L										size N																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", and "A3 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", and "T3 Post-indexed" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<size> Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list> Is a list containing the single 64-bit name of the SIMD&FP register holding the element. The list must be { <Dd>[<index>] }. The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8
 <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.
<size> == 16
 <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.
<size> == 32
 <index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> When <size> == 8, <align> must be omitted, otherwise it is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and the encoding depends on <size>:

<size> == 8
 Encoded in the "index_align<0>" field as 0.
<size> == 16
 Encoded in the "index_align<1:0>" field as 0b00.
<size> == 32
 Encoded in the "index_align<2:0>" field as 0b000.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16
 <align> is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as 0b01.
<size> == 32
 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as 0b011.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_STORE, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    MemU{8*ebytes}(address) = D(d)[index*:(8*ebytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + ebytes;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST2 (multiple 2-element structures)

Store multiple 2-element structures from two or four registers stores multiple 2-element structures from two or four registers to memory, with interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is saved. For details of the addressing mode, see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				1	0	0	x	size		align		Rm							
L										itype																									

Encoding for the Offset variant

Applies when (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				0	0	1	1	size		align		Rm			
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn			Vd			1	0	0	x	size		align		Rm					
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

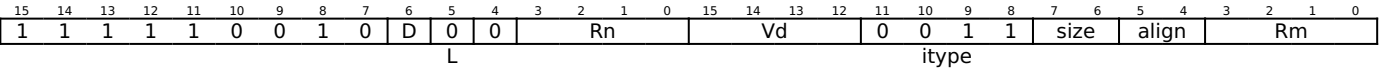
```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST2 \(multiple 2-element structures\)](#).
Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

- <c> For the "A1 Offset", "A1 Post-indexed", "A2 Offset", and "A2 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q> For the "T1 Offset", "T1 Post-indexed", "T2 Offset", and "T2 Post-indexed" variants: see [Standard assembler syntax fields](#).
- <size> See [Standard assembler syntax fields](#).
Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

- { <Dd>, <Dd+1> }
Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1000.
- { <Dd>, <Dd+2> }
Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1001.
- { <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }
Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

- 64
64-bit alignment, encoded in the "align" field as 0b01.
- 128
128-bit alignment, encoded in the "align" field as 0b10.
- 256
256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
        tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to pairs-1 do
        for e = 0 to elements-1 do
            MemU{8*bytes}(address) = D(d+r)[e*:(8*bytes)];
            MemU{8*bytes}(address+bytes) = D(d2+r)[e*:(8*bytes)];
            address = address + 2*bytes;
        end;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 16*pairs;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VST2 (single 2-element structure from one lane)

Store single 2-element structure from one lane of two registers stores one 2-element structure to memory from corresponding elements of two registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) and [A3](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn			Vd			0			0	0	1	index		align		Rm			
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				0		1	0	1	index align		Rm				
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d2 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				1	0	0	1	index_align		Rm					
L											size											N									

Encoding for the Offset variant

Applies when (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d2 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	0	0	1	index_align				Rm			
L												size N																			

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

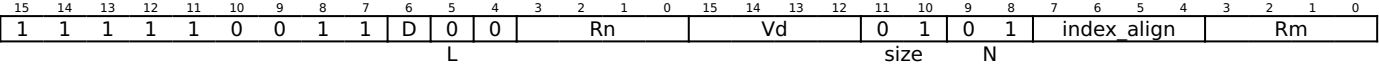
```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

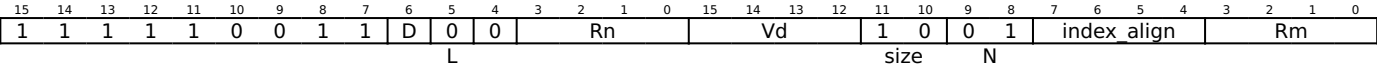
```
if size == '11' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST2 \(single 2-element structure from one lane\)](#).

Assembler Symbols

- <c> For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", and "A3 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q> For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", and "T3 Post-indexed" variants: see [Standard assembler syntax fields](#).
- <size> See [Standard assembler syntax fields](#).
Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list>	Is a list containing the 64-bit names of the two SIMD&FP registers holding the element. The list must be one of: <pre> { <Dd>[<index>], <Dd+1>[<index>] } Single-spaced registers, encoded as "spacing" = 0. { <Dd>[<index>], <Dd+2>[<index>] } Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8. </pre> The encoding of "spacing" depends on <size>: <pre> <size> == 16 "spacing" is encoded in the "index_align<1>" field. <size> == 32 "spacing" is encoded in the "index_align<2>" field. </pre> The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>: <pre> <size> == 8 <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field. <size> == 16 <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field. <size> == 32 <index> is 0 or 1, encoded in the "index_align<3>" field. </pre>
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<align>	Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on <size>: <pre> <size> == 8 Encoded in the "index_align<0>" field as 0. <size> == 16 Encoded in the "index_align<0>" field as 0. <size> == 32 Encoded in the "index_align<1:0>" field as 0b00. </pre> Whenever <align> is present, the permitted values and encoding depend on <size>: <pre> <size> == 8 <align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1. <size> == 16 <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1. <size> == 32 <align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01. </pre> : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.
<Rm>	Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                         tagchecked, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    MemU{8*bytes}(address) = D(d)[index*:(8*bytes)];
    MemU{8*bytes}(address+bytes) = D(d2)[index*:(8*bytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 2*bytes;
        end;
    end;
end;
end;

```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST3 (multiple 3-element structures)

Store multiple 3-element structures from three registers stores multiple 3-element structures to memory from three registers, with interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is saved. For details of the addressing mode, see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPT* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn			Vd			0	1	0	x	size		align		Rm					
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' || align[1] == '1' then Undefined(); end;
var inc : integer;
case itype of
  when '0100' =>
    inc = 1;
  when '0101' =>
    inc = 2;
  otherwise =>
    See("Related encodings");
end;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn			Vd			0	1	0	x	size		align		Rm					
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)


```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' || align[1] == '1' then Undefined(); end;
var inc : integer;
case itype of
  when '0100' =>
    inc = 1;
  when '0101' =>
    inc = 2;
  otherwise =>
    See("Related encodings");
end;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST3 \(multiple 3-element structures\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c>	For the "A1 Offset" and "A1 Post-indexed" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 Offset" and "T1 Post-indexed" variants: see Standard assembler syntax fields .								
<q>	See Standard assembler syntax fields .								
<size>	Is the data size, encoded in "size": <table> <tr> <th>size</th><th><size></th></tr> <tr> <td>00</td><td>8</td></tr> <tr> <td>01</td><td>16</td></tr> <tr> <td>10</td><td>32</td></tr> </table>	size	<size>	00	8	01	16	10	32
size	<size>								
00	8								
01	16								
10	32								
<list>	Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of: { <Dd>, <Dd+1>, <Dd+2> } Single-spaced registers, encoded in the "itype" field as 0b0100. { <Dd>, <Dd+2>, <Dd+4> } Double-spaced registers, encoded in the "itype" field as 0b0101. The register <Dd> is encoded in the "D:Vd" field.								
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.								
<align>	Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access , and is encoded in the "align" field as 0b00. Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as 0b01. : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode .								

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.
For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_STORE, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    for e = 0 to elements-1 do
        MemU{8*bytes}(address          ) = D(d)[e:(8*bytes)];
        MemU{8*bytes}(address+bytes      ) = D(d2)[e:(8*bytes)];
        MemU{8*bytes}(address+2*bytes    ) = D(d3)[e:(8*bytes)];
        address = address + 3*bytes;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 24;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST3 (single 3-element structure from one lane)

Store single 3-element structure from one lane of three registers stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) and [A3](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				0	0	1	0	index_align		Rm					
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d3 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				0	1	1	0	index_align				Rm						
L											size																N							

Encoding for the Offset variant

Applies when (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If **d3 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				1	0	1	0	index align				Rm			
L											size											N									

Encoding for the Offset variant

Applies when (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If **d3 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	0	1	0	index align				Rm							
L												size												N											

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

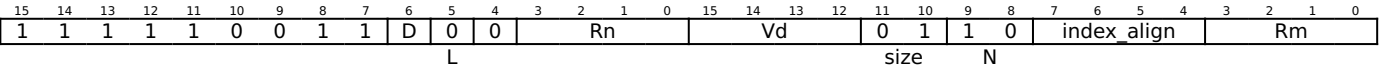
```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d3 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

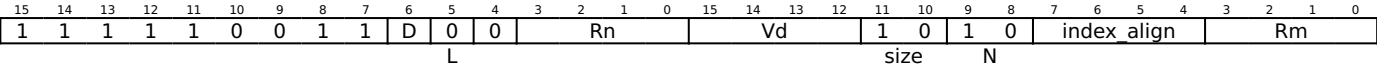
```
if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST3 \(single 3-element structure from one lane\)](#).

Assembler Symbols

`<c>`
`<q>`
`<size>`

For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", and "A3 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", and "T3 Post-indexed" variants: see [Standard assembler syntax fields](#).

See [Standard assembler syntax fields](#).

Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the three SIMD&FP registers holding the element. The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }
Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 8
"spacing" is encoded in the "index_align<0>" field.

<size> == 16
"spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.

<size> == 32
"spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Alignment

Standard alignment rules apply, see [Alignment support](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let address : bits(32) = R(n);
    MemU{8*bytes}(address) = D(d)[index*:(8*bytes)];
    MemU{8*bytes}(address+bytes) = D(d2)[index*:(8*bytes)];
    MemU{8*bytes}(address+2*bytes) = D(d3)[index*:(8*bytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 3*bytes;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST4 (multiple 4-element structures)

Store multiple 4-element structures from four registers stores multiple 4-element structures to memory from four registers, with interleaving. For more information, see [Element and structure load/store instructions](#). Every element of each register is saved. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn			Vd			0			0	0	x	size		align		Rm			
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if itype != '000x' then See("Related encodings"); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				0	0	0	x	size		align		Rm			
L										itype																					

Encoding for the Offset variant

Applies when (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)


```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if itype != '000x' then See("Related encodings"); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST4 \(multiple 4-element structures\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c> For the "A1 Offset" and "A1 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the "T1 Offset" and "T1 Post-indexed" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<size> Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }
Single-spaced registers, encoded in the "itype" field as 0b0000.
{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> }
Double-spaced registers, encoded in the "itype" field as 0b0001.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as 0b01.
128 128-bit alignment, encoded in the "align" field as 0b10.
256 256-bit alignment, encoded in the "align" field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_STORE, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    for e = 0 to elements-1 do
        MemU{8*bytes}(address          ) = D(d)[e:(8*bytes)];
        MemU{8*bytes}(address+bytes     ) = D(d2)[e:(8*bytes)];
        MemU{8*bytes}(address+2*bytes   ) = D(d3)[e:(8*bytes)];
        MemU{8*bytes}(address+3*bytes   ) = D(d4)[e:(8*bytes)];
        address = address + 4*bytes;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 32;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST4 (single 4-element structure from one lane)

Store single 4-element structure from one lane of four registers stores one 4-element structure to memory from corresponding elements of four registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) and [A3](#)) and T32 ([T1](#) , [T2](#) and [T3](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				0		0	1	1	index_align				Rm			
L										size										N												

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '00' then See("Related encodings"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn			Vd			0 1		1	1	index_align			Rm						
L											size										N										

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST4{<c>}{<q>}.size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '01' then See("Related encodings"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d4 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn			Vd			1	0	1	1	index_align			Rm						
L											size											N									

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST4{<c>}{<q>}.size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST4{<c>}{<q>}.size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '10' then See("Related encodings"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d4 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				0	0	1	1	index_align				Rm			
L												size												N							

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if size != '00' then See("Related encodings"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If **d4 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn			Vd			0	1	1	1	index_align			Rm						
L										size										N											

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if size != '01' then See("Related encodings"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If **d4 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				1	0	1	1	index align				Rm											
L												size												N															

Encoding for the Offset variant

Applies when (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if size != '10' then See("Related encodings"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If **d4 > 31**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VST4 (single 4-element structure from one lane)*.

Assembler Symbols

<c> For the "A1 Offset", "A1 Post-indexed", "A2 Offset", "A2 Post-indexed", "A3 Offset", and "A3 Post-indexed" variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1 Offset", "T1 Post-indexed", "T2 Offset", "T2 Post-indexed", "T3 Offset", and "T3 Post-indexed" variants: see [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<size> Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the four SIMD&FP registers holding the element. The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] }
Single-spaced registers, encoded as "spacing" = 0.
{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] }
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 16
"spacing" is encoded in the "index_align<1>" field.
<size> == 32
"spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.
<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.
<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and the encoding depends on <size>:

<size> == 8
Encoded in the "index_align<0>" field as 0.
<size> == 16
Encoded in the "index_align<0>" field as 0.
<size> == 32
Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8
<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.
<size> == 16
<align> is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.
<size> == 32
<align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp\_STORE, nontemporal,
                                                    tagchecked, privileged);

    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    MemU{8*bytes}(address) = D(d)[index*(8*bytes)];
    MemU{8*bytes}(address+bytes) = D(d2)[index*(8*bytes)];
    MemU{8*bytes}(address+2*bytes) = D(d3)[index*(8*bytes)];
    MemU{8*bytes}(address+3*bytes) = D(d4)[index*(8*bytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 4*bytes;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSTM, VSTMDB, VSTMIA

Store multiple SIMD&FP registers stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the alias [VPUSH](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	P	U	D	W	0	Rn				Vd				1	0	1	1	x	x	x	x	x	x	x	0
cond				L								size								imm8											

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist> // (Preferred syntax)
```

```
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				1	1	0	P	U	D	W	0	Rn				Vd				1	0	1	0	imm8											
cond												L												size											

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist> // (Preferred syntax)
```

```
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd:D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	0	Rn				Vd				1	0	1	1	x	x	x	x	x	x	x	0
L												size						imm8													

Encoding for the Decrement Before variant

Applies when `(P == 1 && U == 0 && W == 1)`

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when `(P == 0 && U == 1)`

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist> // (Preferred syntax)
```

```
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIA".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

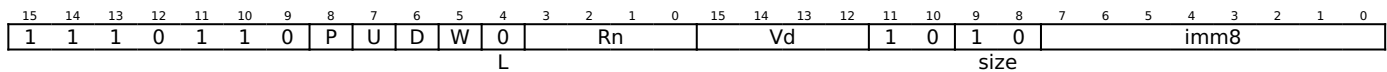
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist> // (Preferred syntax)

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd::D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VSTM](#).

Related encodings: See [Advanced SIMD and floating-point 64-bit move](#) for the T32 instruction set, or [Advanced SIMD and floating-point 64-bit move](#) for the A32 instruction set.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<size>	An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.
<dreglist>	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.
!	Specifies base register writeback, encoded in "W":

W	!
0	[absent]
1	[present]

<reglist>

Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

Alias Conditions

Alias	Is preferred when
VPUSH	P == '1' && U == '0' && W == '1' && Rn == '1101'

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckVFPEEnabled(TRUE);
  var address : bits(32) = if add then R(n) else R(n)-imm32;
  for r = 0 to regs-1 do
    if single_regs then
      MemA{32}(address) = S(d+r);
      address = address+4;
    else
      // Store as two word-aligned words in the correct order for current endianness.
      if BigEndian(AccessType ASIMD) then
        MemA{32}(address) = D(d+r) [63:32];
        MemA{32}(address+4) = D(d+r) [31:0];
      else
        MemA{32}(address) = D(d+r) [31:0];
        MemA{32}(address+4) = D(d+r) [63:32];
      end;
      address = address+8;
    end;
  end;

  if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VSTR

Store SIMD&FP register stores a single register from the Advanced SIMD and floating-point register file to memory, using an address from a general-purpose register, with an optional offset.

Depending on settings in the *CPACR*, *NSACR*, *HCPtr*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	1	U	D	0	0	Rn				Vd				1	0	size		imm8							
cond				P				W				L																			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #(<+/->)<imm>}]
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VSTR{<c>}{<q>}.32 <Sd>, [<Rn>{, #(<+/->)<imm>}]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VSTR{<c>}{<q>}.64 <Dd>, [<Rn>{, #(<+/->)<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	U	D	0	0	Rn				Vd				1	0	size		imm8							
P							W				L																				

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

```
VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #(<+/->)<imm>}]
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VSTR{<c>}{<q>}.32 <Sd>, [<Rn>{, #(<+/->)<imm>}]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #(<+/->)<imm>}]
```

Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<Sd>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vd:D" field.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1"> <thead> <tr> <th>U</th><th>+/-</th></tr> </thead> <tbody> <tr> <td>0</td><td>-</td></tr> <tr> <td>1</td><td>+</td></tr> </tbody> </table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the "A1 Half-precision scalar" and "T1 Half-precision scalar" variants: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2. For the "A1 Double-precision scalar", "A1 Single-precision scalar", "T1 Double-precision scalar", and "T1 Single-precision scalar" variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.						
.32	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.						
.64	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.						
<Dd>	Is the 64-bit name of the SIMD&FP source register, encoded in the "D:Vd" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let address : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    case esize of
        when 16 =>
            MemA{16}(address) = H(d);
        when 32 =>
            MemA{32}(address) = S(d);
        when 64 =>
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType_ASIMD) then
                MemA{32}(address) = D(d) [63:32];
                MemA{32}(address+4) = D(d) [31:0];
            else
                MemA{32}(address) = D(d) [31:0];
                MemA{32}(address+4) = D(d) [63:32];
            end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VSUB (floating-point)

Vector Subtract (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
U												opc								o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

`VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	1	1	Vn				Vd				1	0	size		N	1	M	0	Vm			
cond								o0				o1																o2			

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (size == 01)

`VSUB{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>`

Encoding for the Single-precision scalar variant

Applies when (size == 10)

`VSUB{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when (size == 11)

`VSUB{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>`

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && cond != '1110' then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
U												opc												o1							

Encoding for the 64-bit SIMD vector variant

Applies when (`Q == 0`)

`VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (`Q == 1`)

`VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then Undefined(); end;
if sz == '1' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	0	D	1	1	Vn					Vd					1	0	size	N	1	M	0	Vm			
o0								o1																o2								

Encoding for the Half-precision scalar variant (FEAT_FP16)

Applies when (`size == 01`)

`VSUB{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>`

Encoding for the Single-precision scalar variant

Applies when (`size == 10`)

`VSUB{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when (`size == 11`)

VSUB{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then Undefined(); end;
if size == '01' && InITBlock() then UnpredictableProcedure(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "A2 Double-precision scalar", "A2 Half-precision scalar", and "A2 Single-precision scalar" variants: see Standard assembler syntax fields . For the "T1 128-bit SIMD vector", "T1 64-bit SIMD vector", "T2 Double-precision scalar", "T2 Half-precision scalar", and "T2 Single-precision scalar" variants: see Standard assembler syntax fields .						
<q>	See Standard assembler syntax fields .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table border="1"> <thead> <tr> <th>sz</th><th><dt></th></tr> </thead> <tbody> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </tbody> </table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                D(d+r)[e*esize] = FPSub{esize}(D(n+r)[e*esize], D(m+r)[e*esize], fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 16 =>
                H(d) = FPSub{16}(H(n), H(m), fpcr);
            when 32 =>
                S(d) = FPSub{32}(S(n), S(m), fpcr);
            when 64 =>
                D(d) = FPSub{64}(D(n), D(m), fpcr);
        end;
    end;
end;
```

VSUB (integer)

Vector Subtract (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	0	0	1	1	0	D	size			Vn				Vd				1				0	0	0	N	Q	M	0	Vm			
U										opc										o1															

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			
U										opc										o1											

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields. This encoding must be unconditional.
- <q> For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields. See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = D(n+r)[e*:esize] - D(m+r)[e*:esize];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VSUBHN

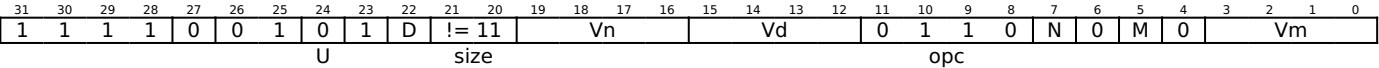
Vector Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are truncated. For rounded results, see [VRSUBHN](#).

There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1



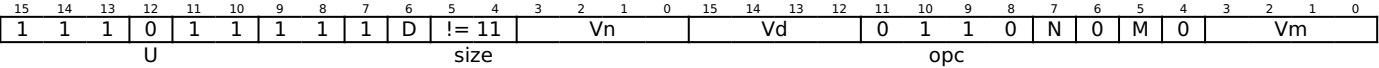
Encoding

VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1



Encoding

VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

- <c>

For the "A1" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- <q>

For the "T1" variant: see [Standard assembler syntax fields](#).
- <dt>

See [Standard assembler syntax fields](#).
- Is the data type for the elements of the operands, encoded in “size”:
- | size | <dt> |
|------|------|
| 00 | I16 |
| 01 | I32 |
| 10 | I64 |
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled\(\);
    for e = 0 to elements-1 do
        let result : bits(2*esize) = Qin(n>>1)[e*:(2*esize)] - Qin(m>>1)[e*:(2*esize)];
        D(d)[e*:esize] = result[2*esize-1:esize];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

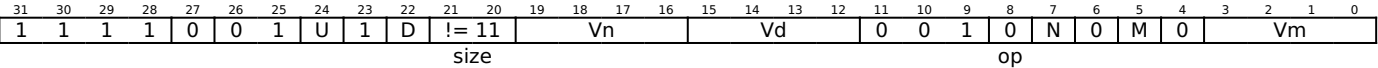
VSUBL

Vector Subtract Long subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in a quadword vector. Before subtracting, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



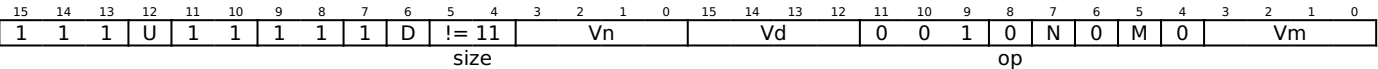
Encoding

VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1



Encoding

VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

- <c>For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.
For the "T1" variant: see *Standard assembler syntax fields*.
- <q>See *Standard assembler syntax fields*.
- <dt>Is the data type for the elements of the second operand vector, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32
- <Qd>Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Dn>Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm>Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vsubw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 - element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

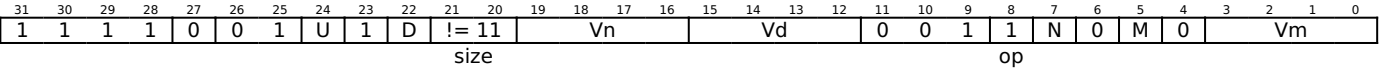
VSUBW

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



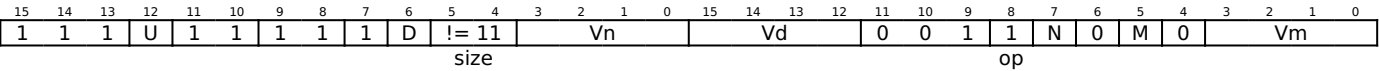
Encoding

```
VSUBW{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

T1



Encoding

```
VSUBW{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>
```

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c>

For the "A1" variant: see *Standard assembler syntax fields*. This encoding must be unconditional.

For the "T1" variant: see *Standard assembler syntax fields*.

<q>

See *Standard assembler syntax fields*.

<dt>

Is the data type for the elements of the second operand vector, encoded in "(U :: size)":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vsubw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 - element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

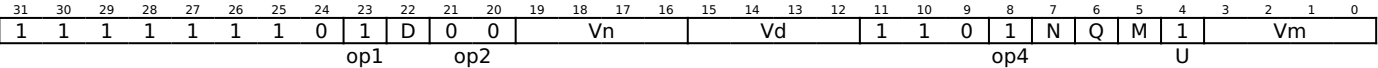
Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSUDOT (by element)

Dot Product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1
(FEAT_AA32I8MM)



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

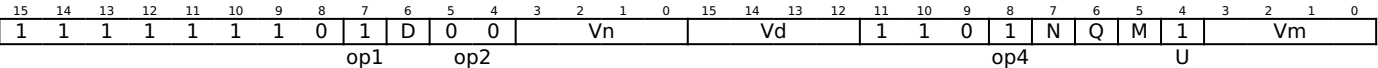
Applies when (Q == 1)

VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let op1_unsigned : boolean = (U == '0');
let op2_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

T1
(FEAT_AA32I8MM)



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let op1_unsigned : boolean = (U == '0');
let op2_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
- <index> Is the element index in the range 0 to 1, encoded in the "M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
 <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```

CheckAdvSIMDEnabled();
var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

operand2 = Din(m);
for r = 0 to regs-1 do
  operand1 = Din(n+r);
  result = Din(d+r);
  for e = 0 to 1 do
    var res : bits(32) = result[e*32];
    for b = 0 to 3 do
      let op1elt : bits(8) = operand1[(4 * e + b)*8];
      let op2elt : bits(8) = operand2[(4 * i + b)*8];
      let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
      let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
      res = res + element1 * element2;
    end;
    result[e*32] = res;
  end;
  D(d+r) = result;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSWP

Vector Swap exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	0	0	1	0	Vd				0	0	0	0	0	0	Q	M	0	Vm			
										size		opc1				opc2																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	0	0	1	0	Vd				0	0	0	0	0	0	Q	M	0	Vm			
										size		opc1				opc2																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c>

For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*. This encoding must be unconditional.
- <q>

For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see *Standard assembler syntax fields*.
- <dt>

See *Standard assembler syntax fields*.
- <Dd>

An optional data type. It is ignored by assemblers, and does not affect the encoding.
- <Dm>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Qd>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- <Qm>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        if d == m then
            D(d+r) = ARBITRARY : bits(64);
        else
            D(d+r) = Din(m+r);
            D(m+r) = Din(d+r);
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VTBL, VTBX

Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0. Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	Vn			Vd			1 0		len	N	op	M	0	Vm						

Encoding for the VTBL variant

Applies when (op == 0)

VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>

Encoding for the VTBX variant

Applies when (op == 1)

VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>

Decode for all variants of this encoding

```
let is_vtbl : boolean = (op == '0');
let length : integer = UInt(len)+1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
if n+length > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n + length > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	Vn			Vd			1	0	len	N	op	M	0	Vm						

Encoding for the VTBL variant

Applies when (op == 0)

VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>

Encoding for the VTBX variant

Applies when (op == 1)

VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>

Decode for all variants of this encoding

```
let is_vtbl : boolean = (op == '0');
let length : integer = UInt(len)+1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
if n+length > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `n + length > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	For the "A1 VTBL" and "A1 VTBX" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 VTBL" and "T1 VTBX" variants: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<list>	The vectors containing the table. It must be one of: <Dn> Encoded as len = 0b00. <Dn>, <Dn+1> Encoded as len = 0b01. <Dn>, <Dn+1>, <Dn+2> Encoded as len = 0b10. <Dn>, <Dn+1>, <Dn+2>, <Dn+3> Encoded as len = 0b11.
<Dm>	Is the 64-bit name of the SIMD&FP source register holding the indices, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
    let table3 : bits(64) = if length == 4 then D(n+3) else Zeros{64};
    let table2 : bits(64) = if length >= 3 then D(n+2) else Zeros{64};
    let table1 : bits(64) = if length >= 2 then D(n+1) else Zeros{64};
    let table : bits(256) = table3 :: table2 :: table1 :: D(n);

    for i = 0 to 7 do
        let index : integer = UInt(D(m)[i*:8]);
        if index < 8*length then
            D(d)[i*:8] = table[index*:8];
        else
            if is_vtbl then
                D(d)[i*:8] = Zeros{8};
                // else D[d]<i*:8> unchanged
            end;
        end;
    end;
end;
```

Operational information

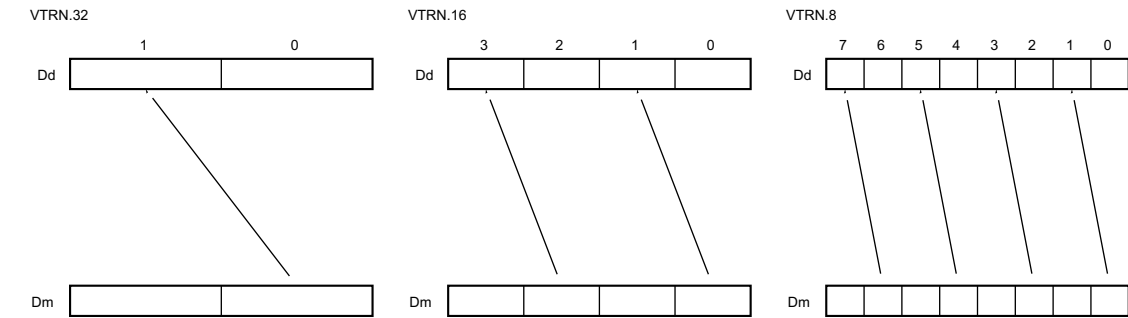
This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VTRN

Vector Transpose treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices. The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types. The following figure shows an example of the operation of VTRN doubleword operations.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*. This instruction is used by the pseudo-instructions *VUZP (alias)*, and *VZIP (alias)*. It has encodings from the following instruction sets: A32 (*A1*) and T32 (*T1*) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd					0	0	0	0	1	Q	M	0	Vm			
opc1														opc2																		

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd				0	0	0	0	1	Q	M	0	Vm			
opc1														opc2																		

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```


Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.										
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .										
<dt>	See Standard assembler syntax fields .										
	Is the data type for the elements of the vectors, encoded in "size":										
<table><tr><th>size</th><th><dt></th></tr><tr><td>00</td><td>8</td></tr><tr><td>01</td><td>16</td></tr><tr><td>10</td><td>32</td></tr><tr><td>11</td><td>RESERVED</td></tr></table>		size	<dt>	00	8	01	16	10	32	11	RESERVED
size	<dt>										
00	8										
01	16										
10	32										
11	RESERVED										
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.										
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.										
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.										
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let h : integer = elements DIV 2;

    for r = 0 to regs-1 do
        if d == m then
            D(d+r) = ARBITRARY : bits(64);
        else
            for e = 0 to h-1 do
                D(d+r) [(2*e+1)*:esize] = Din(m+r) [(2*e)*:esize];
                D(m+r) [(2*e)*:esize] = Din(d+r) [(2*e+1)*:esize];
            end;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VTST

Vector Test Bits takes each element in a vector, and bitwise ANDs it with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit fields.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size	Vn				Vd				1	0	0	0	N	Q	M	1	Vm				
U											opc											o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VTST{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VTST{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size	Vn				Vd				1	0	0	0	N	Q	M	1	Vm				
U										opc												o1									

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VTST{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VTST{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.
<q>	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields . See Standard assembler syntax fields .

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	8
01	16
10	32

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if !IsZero(D(n+r)[e*:esize] AND D(m+r)[e*:esize]) then
                D(d+r)[e*:esize] = Ones(esize);
            else
                D(d+r)[e*:esize] = Zeros(esize);
            end;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in *About the DIT bit*.

VUDOT (by element)

Dot Product index form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1
(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	1	0	Vn			Vd			1			1	0	1	N	Q	M	1	Vm			
op1									op2			op4									U										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let signed : boolean = (U=='0');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm[3:0]);
let index : integer = UInt(M);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

T1
(FEAT_DotProd)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	1	0	Vn			Vd			1	1	0	1	N	Q	M	1	Vm					
op1									op2			op4									U										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let signed : boolean = (U=='0');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm[3:0]);
let index : integer = UInt(M);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```

var operand1 : bits(64);
let operand2 : bits(64) = D(m);
var result : bits(64);
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    operand1 = D(n+r);
    result = D(d+r);
    var element1, element2 : integer;
    for e = 0 to 1 do
        var res : integer = 0;
        for i = 0 to 3 do
            if signed then
                element1 = SInt(operand1[(4 * e + i):(esize DIV 4)]);
                element2 = SInt(operand2[(4 * index + i):(esize DIV 4)]);
            else
                element1 = UInt(operand1[(4 * e + i):(esize DIV 4)]);
                element2 = UInt(operand2[(4 * index + i):(esize DIV 4)]);
            end;
            res = res + element1 * element2;
        end;
        result[e*esize] = result[e*esize] + res;
    end;
    D(d+r) = result;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUDOT (vector)

Dot Product vector form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an `OPTIONAL` instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

`ID_ISAR6`.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	0	Vn			Vd			1	1	0	1	N	Q	M	1	Vm					
op1							op2				op3				op4				U												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let signed : boolean = U=='0';
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_DotProd)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	0	Vn			Vd			1	1	0	1	N	Q	M	1	Vm					
op1							op2				op3				op4				U												

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_DotProd) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let signed : boolean = U=='0';
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 32;
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```

var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);
CheckAdvSIMDEnabled();
for r = 0 to regs-1 do
    operand1 = D(n+r);
    operand2 = D(m+r);
    result = D(d+r);
    var element1, element2 : integer;
    for e = 0 to 1 do
        var res : integer = 0;
        for i = 0 to 3 do
            if signed then
                element1 = SInt(operand1[(4 * e + i):(esize DIV 4)]);
                element2 = SInt(operand2[(4 * e + i):(esize DIV 4)]);
            else
                element1 = UInt(operand1[(4 * e + i):(esize DIV 4)]);
                element2 = UInt(operand2[(4 * e + i):(esize DIV 4)]);
            end;
            res = res + element1 * element2;
        end;
        result[e*esize] = result[e*esize] + res;
    end;
    D(d+r) = result;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

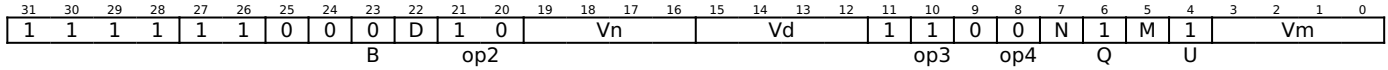
VUMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32I8MM)



Encoding

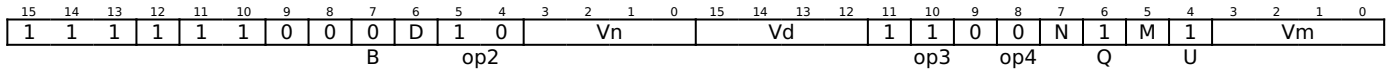
VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
  when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1

(FEAT_AA32I8MM)



Encoding

VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
  when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .
<Qd>	Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();  
let operand1 : bits(128) = Q(n>>1);  
let operand2 : bits(128) = Q(m>>1);  
let addend : bits(128)   = Q(d>>1);  
  
Q(d>>1) = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

2026-03_rel 2026-03-26 20:48:11

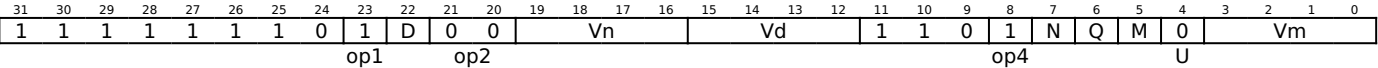
Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUSDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1
(FEAT_AA32I8MM)



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

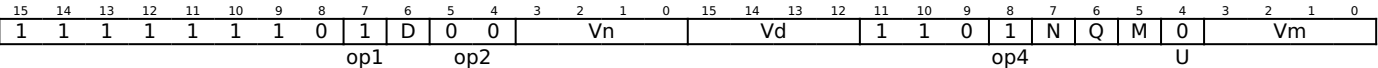
Applies when (Q == 1)

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let op1_unsigned : boolean = (U == '0');
let op2_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

T1
(FEAT_AA32I8MM)



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let op1_unsigned : boolean = (U == '0');
let op2_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(Vm);
let i : integer = UInt(M);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
- <index> Is the element index in the range 0 to 1, encoded in the "M" field.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
 <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Operation

```

CheckAdvSIMDEnabled();
var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

operand2 = Din(m);
for r = 0 to regs-1 do
  operand1 = Din(n+r);
  result = Din(d+r);
  for e = 0 to 1 do
    var res : bits(32) = result[e*32];
    for b = 0 to 3 do
      let op1elt : bits(8) = operand1[(4 * e + b)*8];
      let op2elt : bits(8) = operand2[(4 * i + b)*8];
      let element1 : integer = if op1_unsigned then UInt(op1elt) else SInt(op1elt);
      let element2 : integer = if op2_unsigned then UInt(op2elt) else SInt(op2elt);
      res = res + element1 * element2;
    end;
    result[e*32] = res;
  end;
  D(d+r) = result;
end;

```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUSDOT (vector)

Dot Product vector form with mixed-sign integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	1	D	1	0	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
op1								op2				op3								op4				U							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32I8MM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	1	D	1	0	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
op1								op2				op3								op4				U							

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See <i>Standard assembler syntax fields</i> .
<Dd>	Is the 64-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Qd>	Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
var operand1 : bits(64);
var operand2 : bits(64);
var result : bits(64);

for r = 0 to regs-1 do
  operand1 = Din(n+r);
  operand2 = Din(m+r);
  result = Din(d+r);
  for e = 0 to 1 do
    var res : bits(32) = result[e*32];
    for b = 0 to 3 do
      let element1 : integer = UInt(operand1[(4 * e + b)*8]);
      let element2 : integer = SInt(operand2[(4 * e + b)*8]);
      res = res + element1 * element2;
    end;
    result[e*32] = res;
  end;
  D(d+r) = result;
end;
```

2026-03_rel 2026-03-26 20:48:11

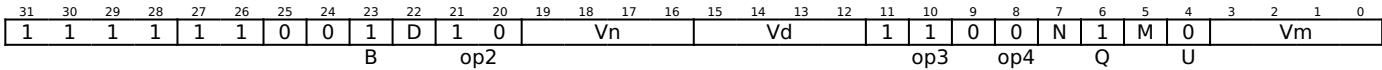
Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUSMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1 (FEAT_AA32I8MM)



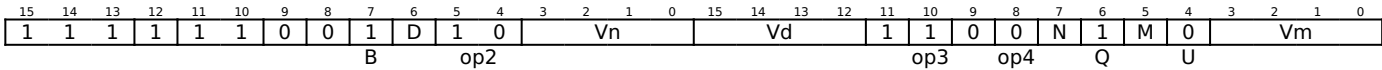
Encoding

VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
  when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1 (FEAT_AA32I8MM)



Encoding

VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AA32I8MM) then Undefined(); end;
var op1_unsigned : boolean;
var op2_unsigned : boolean;
case B::U of
  when '00' => op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' => op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' => op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' => Undefined();
end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <Qd> Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();  
let operand1 : bits(128) = Q(n>>1);  
let operand2 : bits(128) = Q(m>>1);  
let addend : bits(128)   = Q(d>>1);  
  
Q(d>>1) = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUZP

Vector Unzip de-interleaves the elements of two vectors.
The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.
The following figure shows an example of the operation of VUZP doubleword operation for data type 8.

VUZP.8, doubleword																
	Register state before operation								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B6	B4	B2	B0	A6	A4	A2	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	B5	B3	B1	A7	A5	A3	A1

The following figure shows an example of the operation of VUZP quadword operation for data type 32.

VUZP.32, quadword								
	Register state before operation				Register state after operation			
Qd	A3	A2	A1	A0	B2	B0	A2	A0
Qm	B3	B2	B1	B0	B3	B1	A3	A1

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	0	0	1	1	1	D	1	1	size		opc1		Vd					0	0	0	1	0	Q	M	0	Vm				

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd				0	0	0	1	0	Q	M	0	Vm			
opc1															opc2																	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```


Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .																		
<q>	See Standard assembler syntax fields .																		
<dt>	For the "A1 64-bit SIMD vector" and "T1 64-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "size": <table><tr><th>size</th><th><dt></th></tr><tr><td>00</td><td>8</td></tr><tr><td>01</td><td>16</td></tr><tr><td>1x</td><td>RESERVED</td></tr></table> For the "A1 128-bit SIMD vector" and "T1 128-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "size": <table><tr><th>size</th><th><dt></th></tr><tr><td>00</td><td>8</td></tr><tr><td>01</td><td>16</td></tr><tr><td>10</td><td>32</td></tr><tr><td>11</td><td>RESERVED</td></tr></table>	size	<dt>	00	8	01	16	1x	RESERVED	size	<dt>	00	8	01	16	10	32	11	RESERVED
size	<dt>																		
00	8																		
01	16																		
1x	RESERVED																		
size	<dt>																		
00	8																		
01	16																		
10	32																		
11	RESERVED																		
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.																		
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.																		
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.																		
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.																		

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q(d>>1) = ARBITRARY : bits(128);
        else
            let zipped_q : bits(256) = Q(m>>1)::Q(d>>1);
            for e = 0 to (128 DIV esize) - 1 do
                Q(d>>1)[e*:esize] = zipped_q[(2*e)*:esize];
                Q(m>>1)[e*:esize] = zipped_q[(2*e+1)*:esize];
            end;
        end;
    else
        if d == m then
            D(d) = ARBITRARY : bits(64);
        else
            let zipped_d : bits(128) = D(m)::D(d);
            for e = 0 to (64 DIV esize) - 1 do
                D(d)[e*:esize] = zipped_d[(2*e)*:esize];
                D(m)[e*:esize] = zipped_d[(2*e+1)*:esize];
            end;
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VUZP (alias)

Vector Unzip de-interleaves the elements of two vectors.

This is a pseudo-instruction of [VTRN](#). This means:

- The encodings in this description are named to match the encodings of [VTRN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VTRN](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	1	0	1	0	Vd				0	0	0	0	1	0	M	0	Vm			
size												opc1				opc2				Q											

Encoding for the 64-bit SIMD vector variant

VUZP{<c>} {<q>} .32 <Dd>, <Dm>

is equivalent to

VTRN{<c>} {<q>} .<dt> <Dd>, <Dm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	1	0	1	0	Vd				0	0	0	0	1	0	M	0	Vm			
size												opc1				opc2				Q											

Encoding for the 64-bit SIMD vector variant

VUZP{<c>} {<q>} .32 <Dd>, <Dm>

is equivalent to

VTRN{<c>} {<q>} .<dt> <Dd>, <Dm>

Assembler Symbols

<c>	For the "A1 64-bit SIMD vector" variant: see Standard assembler syntax fields . This encoding must be unconditional. For the "T1 64-bit SIMD vector" variant: see Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

The description of [VTRN](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VTRN](#) gives the operational information for this instruction.

VZIP

Vector Zip interleaves the elements of two vectors.
The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.
The following figure shows an example of the operation of VZIP doubleword operation for data type 8.

VZIP.8, doubleword																
	Register state before operation								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B3	A3	B2	A2	B1	A1	B0	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	A7	B6	A6	B5	A5	B4	A4

The following figure shows an example of the operation of VZIP quadword operation for data type 32.

VZIP.32, quadword								
	Register state before operation				Register state after operation			
Qd	A3	A2	A1	A0	B1	A1	B0	A0
Qm	B3	B2	B1	B0	B3	A3	B2	A2

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.
It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd				0	0	0	1	1	Q	M	0	Vm					
														opc1				opc2														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	size			1	0	Vd				0	0	0	1	1	Q	M	0	Vm			
												opc1				opc2																

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>	For the "A1 128-bit SIMD vector" and "A1 64-bit SIMD vector" variants: see Standard assembler syntax fields . This encoding must be unconditional.										
	For the "T1 128-bit SIMD vector" and "T1 64-bit SIMD vector" variants: see Standard assembler syntax fields .										
<q>	See Standard assembler syntax fields .										
<dt>	For the "A1 64-bit SIMD vector" and "T1 64-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "size": <table><tr><th>size</th><th><dt></th></tr><tr><td>00</td><td>8</td></tr><tr><td>01</td><td>16</td></tr><tr><td>1x</td><td>RESERVED</td></tr></table>	size	<dt>	00	8	01	16	1x	RESERVED		
size	<dt>										
00	8										
01	16										
1x	RESERVED										
	For the "A1 128-bit SIMD vector" and "T1 128-bit SIMD vector" variants: is the data type for the elements of the vectors, encoded in "size": <table><tr><th>size</th><th><dt></th></tr><tr><td>00</td><td>8</td></tr><tr><td>01</td><td>16</td></tr><tr><td>10</td><td>32</td></tr><tr><td>11</td><td>RESERVED</td></tr></table>	size	<dt>	00	8	01	16	10	32	11	RESERVED
size	<dt>										
00	8										
01	16										
10	32										
11	RESERVED										
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.										
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.										
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.										
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.										

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q(d>>1) = ARBITRARY : bits(128);
        else
            var zipped_q : bits(256);
            for e = 0 to (128 DIV esize) - 1 do
                zipped_q[(2*e)*:esize] = Q(d>>1)[e*:esize];
                zipped_q[(2*e+1)*:esize] = Q(m>>1)[e*:esize];
            end;
            Q(d>>1) = zipped_q[127:0]; Q(m>>1) = zipped_q[255:128];
        end;
    else
        if d == m then
            D(d) = ARBITRARY : bits(64);
        else
            var zipped_d : bits(128);
            for e = 0 to (64 DIV esize) - 1 do
                zipped_d[(2*e)*:esize] = D(d)[e*:esize];
                zipped_d[(2*e+1)*:esize] = D(m)[e*:esize];
            end;
            D(d) = zipped_d[63:0]; D(m) = zipped_d[127:64];
        end;
    end;
end;
```

Operational information

This instruction is a data-independent-time instruction as described in [About the DIT bit](#).

VZIP (alias)

Vector Zip interleaves the elements of two vectors.

This is a pseudo-instruction of [VTRN](#). This means:

- The encodings in this description are named to match the encodings of [VTRN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VTRN](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	1	0	1	0	Vd				0	0	0	0	1	0	M	0	Vm			
size												opc1				opc2				Q											

Encoding for the 64-bit SIMD vector variant

VZIP{<c>} {<q>} .32 <Dd>, <Dm>

is equivalent to

VTRN{<c>} {<q>} .<dt> <Dd>, <Dm>

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	1	0	1	0	Vd				0	0	0	0	1	0	M	0	Vm			
size												opc1				opc2				Q											

Encoding for the 64-bit SIMD vector variant

VZIP{<c>} {<q>} .32 <Dd>, <Dm>

is equivalent to

VTRN{<c>} {<q>} .<dt> <Dd>, <Dm>

Assembler Symbols

- <c>

For the "A1 64-bit SIMD vector" variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the "T1 64-bit SIMD vector" variant: see [Standard assembler syntax fields](#).
- <q>

See [Standard assembler syntax fields](#).
- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

The description of [VTRN](#) gives the operational pseudocode for this instruction.

Operational Information

The description of [VTRN](#) gives the operational information for this instruction.

A32 instruction set encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				op0																								op1			

Decode fields			Instruction details
cond	op0	op1	
!= 1111	00x		Data-processing and miscellaneous instructions
!= 1111	010		Load/Store Word, Unsigned Byte (immediate, literal)
!= 1111	011	0	Load/Store Word, Unsigned Byte (register)
!= 1111	011	1	Media instructions
	10x		Branch, branch with link, and block data transfer
	11x		System register access, Advanced SIMD, floating-point, and Supervisor call
1111	0xx		Unconditional instructions

Data-processing and miscellaneous instructions

The encodings in this section are decoded from [A32 instruction set encoding](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	op0	op1																		op2	op3	op4				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields					Instruction details
op0	op1	op2	op3	op4	
0	0xxxx	1	00	1	Multiply and Accumulate
0	10xx0	0			Miscellaneous
0	10xx0	1		0	Halfword Multiply and Accumulate
0	1xxxx	1	00	1	Synchronization primitives and Load-Acquire/Store-Release
0		1	!= 00	1	Extra load/store
0	!= 10xx0	0		1	Data-processing register (register shift)
0	!= 10xx0			0	Data-processing register (immediate shift)
1					Data-processing immediate

Multiply and Accumulate

The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	0	0	0	opc				S	RdHi				RdLo				Rm				1	0	0	1	Rn			
cond																																

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details
opc	S	
000	0	MUL, MULS — Not flag setting
000	1	MUL, MULS — Flag setting
001	0	MLA, MLAS — Not flag setting
001	1	MLA, MLAS — Flag setting
01x	1	UNALLOCATED
010	0	UMAAL
011	0	MLS
100	0	UMULL, UMULLS — Not flag setting
100	1	UMULL, UMULLS — Flag setting
101	0	UMLAL, UMLALS — Not flag setting
101	1	UMLAL, UMLALS — Flag setting
110	0	SMULL, SMULLS — Not flag setting
110	1	SMULL, SMULLS — Flag setting
111	0	SMLAL, SMLALS — Not flag setting
111	1	SMLAL, SMLALS — Flag setting

Miscellaneous

The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	1	0	opc0	0																0		op1						
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction details
op0	op1	
00	010	UNALLOCATED
00	011	UNALLOCATED
01	001	BX
01	010	BXJ
01	011	BLX (register)
11	001	CLZ
11	110	ERET
1x	01x	UNALLOCATED
x0	001	UNALLOCATED
	000	Move special register (register)
	100	Cyclic Redundancy Check
	101	Integer Saturating Arithmetic
	111	Exception Generation
!= 11	110	UNALLOCATED

Move special register (register)

The encodings in this section are decoded from [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	opc		0	mask				Rd				(0) (0)		B	m	0	0	0	0	Rn			
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

Instruction Details

opc	B	
x0	0	MRS
x0	1	MRS (Banked register)
x1	0	MSR (register)
x1	1	MSR (Banked register)

Cyclic Redundancy Check

The encodings in this section are decoded from [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	sz		0	Rn				Rd				(0)	(0)	C	(0)	0	1	0	0	Rm			
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

Instruction Details

Condition

sz	C		
00	0	CRC32 — CRC32B	FEAT_CRC32
00	1	CRC32C — CRC32CB	FEAT_CRC32
01	0	CRC32 — CRC32H	FEAT_CRC32
01	1	CRC32C — CRC32CH	FEAT_CRC32
10	0	CRC32 — CRC32W	FEAT_CRC32
10	1	CRC32C — CRC32CW	FEAT_CRC32
11		CONSTRAINED UNPREDICTABLE	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

Integer Saturating Arithmetic

The encodings in this section are decoded from [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	opc		0	Rn				Rd				(0)	(0)	(0)	(0)	0	1	0	1	Rm			
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

Instruction Details

opc	
00	QADD
01	QSUB
10	QDADD
11	QDSUB

Exception Generation

The encodings in this section are decoded from [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	opc		0	imm12											0	1	1	1	imm4				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields	Instruction Details
opc	
00	HLT
01	BKPT
10	HVC
11	SMC

Halfword Multiply and Accumulate

The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	opc			0	Rd			Ra			Rm			1	M	N	0	Rn					
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields			Instruction Details
opc	M	N	
00	0	0	SMLABB, SMLABT, SMLATB, SMLATT — SMLABB
00	0	1	SMLABB, SMLABT, SMLATB, SMLATT — SMLATB
00	1	0	SMLABB, SMLABT, SMLATB, SMLATT — SMLABT
00	1	1	SMLABB, SMLABT, SMLATB, SMLATT — SMLATT
01	0	0	SMLAWB, SMLAWT — SMLAWB
01	0	1	SMULWB, SMULWT — SMULWB
01	1	0	SMLAWB, SMLAWT — SMLAWT
01	1	1	SMULWB, SMULWT — SMULWT
10	0	0	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBB
10	0	1	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTB
10	1	0	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBT
10	1	1	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTT
11	0	0	SMULBB, SMULBT, SMULTB, SMULTT — SMULBB
11	0	1	SMULBB, SMULBT, SMULTB, SMULTT — SMULTB
11	1	0	SMULBB, SMULBT, SMULTB, SMULTT — SMULBT
11	1	1	SMULBB, SMULBT, SMULTB, SMULTT — SMULTT

Synchronization primitives and Load-Acquire/Store-Release

The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	1	op0																			1	0	0	1				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields	Instruction details
op0	
0	UNALLOCATED
1	Load/Store Exclusive and Load-Acquire/Store-Release

Load/Store Exclusive and Load-Acquire/Store-Release

The encodings in this section are decoded from [Synchronization primitives and Load-Acquire/Store-Release](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	size		L	Rn				xRd			(1) (1)		ex	ord	1	0	0	1	xRt				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields				Instruction Details
size	L	ex	ord	
		0	1	UNALLOCATED
00	0	0	0	STL
00	0	1	0	STLEX
00	0	1	1	STREX
00	1	0	0	LDA
00	1	1	0	LDAEX
00	1	1	1	LDREX
01		0	0	UNALLOCATED
01	0	1	0	STLEXD
01	0	1	1	STREXD
01	1	1	0	LDAEXD
01	1	1	1	LDREXD
10	0	0	0	STLB
10	0	1	0	STLEXB
10	0	1	1	STREXB
10	1	0	0	LDAB
10	1	1	0	LDAEXB
10	1	1	1	LDREXB
11	0	0	0	STLH
11	0	1	0	STLEXH
11	0	1	1	STREXH
11	1	0	0	LDAH
11	1	1	0	LDAEXH
11	1	1	1	LDREXH

Extra load/store

The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0	0	0		op0																1	!= 00		1								
cond																												op3							

The following constraints also apply to this encoding: cond != '1111' && op3 != '00'

Decode fields	Instruction details	
op0		
0	Load/Store Dual, Half, Signed Byte (register)	
1	Load/Store Dual, Half, Signed Byte (immediate, literal)	

Load/Store Dual, Half, Signed Byte (register)

The encodings in this section are decoded from [Extra load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
!= 1111				0	0	0	P	U	0	W	o1	Rn				Rt				(0) (0) (0) (0)				1	!= 00				1	Rm									
cond																												op2											

The following constraints also apply to this encoding: cond != '1111' && op2 != '00'

Decode fields				Instruction Details
P	W	o1	op2	
0	0	0	01	STRH (register) — Post-indexed
0	0	0	10	LDRD (register) — Post-indexed
0	0	0	11	STRD (register) — Post-indexed
0	0	1	01	LDRH (register) — Post-indexed
0	0	1	10	LDRSB (register) — Post-indexed
0	0	1	11	LDRSH (register) — Post-indexed
0	1	0	01	STRHT
0	1	0	1x	UNALLOCATED
0	1	1	01	LDRHT
0	1	1	10	LDRSBT
0	1	1	11	LDRSHT
1	0	0	01	STRH (register) — Offset
1	0	0	10	LDRD (register) — Offset
1	0	0	11	STRD (register) — Offset
1	0	1	01	LDRH (register) — Offset
1	0	1	10	LDRSB (register) — Offset
1	0	1	11	LDRSH (register) — Offset
1	1	0	01	STRH (register) — Pre-indexed
1	1	0	10	LDRD (register) — Pre-indexed
1	1	0	11	STRD (register) — Pre-indexed
1	1	1	01	LDRH (register) — Pre-indexed
1	1	1	10	LDRSB (register) — Pre-indexed
1	1	1	11	LDRSH (register) — Pre-indexed

Load/Store Dual, Half, Signed Byte (immediate, literal)

The encodings in this section are decoded from [Extra load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	o1	Rn				Rt				imm4H				1	!= 00	1	imm4L				
cond												op2																			

The following constraints also apply to this encoding: cond != '1111' && op2 != '00'

P:W	Decode fields			Instruction Details
	o1	Rn	op2	
	0	1 1 1 1	1 0	LDRD (literal)
0 0	0		0 1	STRH (immediate) — Post-indexed
0 0	0		1 1	STRD (immediate) — Post-indexed
0 0	0	!= 1 1 1 1	1 0	LDRD (immediate) — Post-indexed
0 0	1	!= 1 1 1 1	0 1	LDRH (immediate) — Post-indexed
0 0	1	!= 1 1 1 1	1 0	LDRSB (immediate) — Post-indexed
0 0	1	!= 1 1 1 1	1 1	LDRSH (immediate) — Post-indexed
0 1	0		0 1	STRHT
0 1	0	1 1 1 1	1 1	UNALLOCATED
0 1	0	!= 1 1 1 1	1 x	UNALLOCATED
0 1	1		0 1	LDRHT
0 1	1		1 0	LDRSBT
0 1	1		1 1	LDRSHT
1 0	0		0 1	STRH (immediate) — Offset
1 0	0		1 1	STRD (immediate) — Offset
1 0	0	!= 1 1 1 1	1 0	LDRD (immediate) — Offset
1 0	1	!= 1 1 1 1	0 1	LDRH (immediate) — Offset
1 0	1	!= 1 1 1 1	1 0	LDRSB (immediate) — Offset
1 0	1	!= 1 1 1 1	1 1	LDRSH (immediate) — Offset
1 1	0		0 1	STRH (immediate) — Pre-indexed
1 1	0		1 1	STRD (immediate) — Pre-indexed
1 1	0	!= 1 1 1 1	1 0	LDRD (immediate) — Pre-indexed
1 1	1	!= 1 1 1 1	0 1	LDRH (immediate) — Pre-indexed
1 1	1	!= 1 1 1 1	1 0	LDRSB (immediate) — Pre-indexed
1 1	1	!= 1 1 1 1	1 1	LDRSH (immediate) — Pre-indexed
!= 0 1	1	1 1 1 1	0 1	LDRH (literal)
!= 0 1	1	1 1 1 1	1 0	LDRSB (literal)
!= 0 1	1	1 1 1 1	1 1	LDRSH (literal)

Data-processing register (register shift)

The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	op0							op1													0			1				
cond																															

The following constraints also apply to this encoding: cond != '1111' && !(op0 == '10' && op1 == '0')

Decode fields		Instruction details
op0	op1	
0 x		Integer Data Processing (three register, register shift)
1 0	1	Integer Test and Compare (two register, register shift)
1 1		Logical Arithmetic (three register, register shift)

Integer Data Processing (three register, register shift)

The encodings in this section are decoded from [Data-processing register \(register shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	opc			S	Rn				Rd				Rs				0	stype		1	Rm			
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details
opc	S	
000	0	AND, ANDS (register-shifted register) — Not flag setting
000	1	AND, ANDS (register-shifted register) — Flag setting
001	0	EOR, EORS (register-shifted register) — Not flag setting
001	1	EOR, EORS (register-shifted register) — Flag setting
010	0	SUB, SUBS (register-shifted register) — Not flag setting
010	1	SUB, SUBS (register-shifted register) — Flag setting
011	0	RSB, RSBS (register-shifted register) — Not flag setting
011	1	RSB, RSBS (register-shifted register) — Flag setting
100	0	ADD, ADDS (register-shifted register) — Not flag setting
100	1	ADD, ADDS (register-shifted register) — Flag setting
101	0	ADC, ADCS (register-shifted register) — Not flag setting
101	1	ADC, ADCS (register-shifted register) — Flag setting
110	0	SBC, SBCS (register-shifted register) — Not flag setting
110	1	SBC, SBCS (register-shifted register) — Flag setting
111	0	RSC, RSCS (register-shifted register) — Not flag setting
111	1	RSC, RSCS (register-shifted register) — Flag setting

Integer Test and Compare (two register, register shift)

The encodings in this section are decoded from [Data-processing register \(register shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	opc		1	Rn				(0)	(0)	(0)	(0)	Rs				0	stype	1	Rm				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details
opc		
00		TST (register-shifted register)
01		TEQ (register-shifted register)
10		CMP (register-shifted register)
11		CMN (register-shifted register)

Logical Arithmetic (three register, register shift)

The encodings in this section are decoded from [Data-processing register \(register shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	opc		S	Rn				Rd				Rs				0	stype	1	Rm				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details
opc	S	
00	0	ORR, ORRS (register-shifted register) — Not flag setting
00	1	ORR, ORRS (register-shifted register) — Flag setting
01	0	MOV, MOVS (register-shifted register) — Not flag setting
01	1	MOV, MOVS (register-shifted register) — Flag setting
10	0	BIC, BICS (register-shifted register) — Not flag setting
10	1	BIC, BICS (register-shifted register) — Flag setting
11	0	MVN, MVNS (register-shifted register) — Not flag setting
11	1	MVN, MVNS (register-shifted register) — Flag setting

Data-processing register (immediate shift)

The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	0	op0						op1															0					
cond																															

The following constraints also apply to this encoding: cond != '1111' && !(op0 == '10' && op1 == '0')

Decode fields		Instruction details
op0	op1	
0x		Integer Data Processing (three register, immediate shift)
10	1	Integer Test and Compare (two register, immediate shift)
11		Logical Arithmetic (three register, immediate shift)

Integer Data Processing (three register, immediate shift)

The encodings in this section are decoded from [Data-processing register \(immediate shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	0	opc			S	Rn				Rd				imm5				styp		0	Rm				
cond																															
								Rn				Rd				imm5				styp				0				Rm			

The following constraints also apply to this encoding: cond != '1111'

Decode fields				Instruction Details
opc	S	Rn	imm5:stype	
000	0		0000011	AND, ANDS (register) — AND, rotate right with extend
000	0		!= 0000011	AND, ANDS (register) — AND, shift or rotate by value
000	1		0000011	AND, ANDS (register) — ANDS, rotate right with extend
000	1		!= 0000011	AND, ANDS (register) — ANDS, shift or rotate by value
001	0		0000011	EOR, EORS (register) — EOR, rotate right with extend
001	0		!= 0000011	EOR, EORS (register) — EOR, shift or rotate by value
001	1		0000011	EOR, EORS (register) — EORS, rotate right with extend
001	1		!= 0000011	EOR, EORS (register) — EORS, shift or rotate by value
010	0	1101	0000011	SUB, SUBS (SP minus register) — SUB, rotate right with extend
010	0	1101	!= 0000011	SUB, SUBS (SP minus register) — SUB, shift or rotate by value
010	0	!= 1101	0000011	SUB, SUBS (register) — SUB, rotate right with extend
010	0	!= 1101	!= 0000011	SUB, SUBS (register) — SUB, shift or rotate by value
010	1	1101	0000011	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
010	1	1101	!= 0000011	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
010	1	!= 1101	0000011	SUB, SUBS (register) — SUBS, rotate right with extend
010	1	!= 1101	!= 0000011	SUB, SUBS (register) — SUBS, shift or rotate by value
011	0		0000011	RSB, RSBS (register) — RSB, rotate right with extend
011	0		!= 0000011	RSB, RSBS (register) — RSB, shift or rotate by value
011	1		0000011	RSB, RSBS (register) — RSBS, rotate right with extend
011	1		!= 0000011	RSB, RSBS (register) — RSBS, shift or rotate by value
100	0	1101	0000011	ADD, ADDS (SP plus register) — ADD, rotate right with extend
100	0	1101	!= 0000011	ADD, ADDS (SP plus register) — ADD, shift or rotate by value
100	0	!= 1101	0000011	ADD, ADDS (register) — ADD, rotate right with extend
100	0	!= 1101	!= 0000011	ADD, ADDS (register) — ADD, shift or rotate by value
100	1	1101	0000011	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
100	1	1101	!= 0000011	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value
100	1	!= 1101	0000011	ADD, ADDS (register) — ADDS, rotate right with extend
100	1	!= 1101	!= 0000011	ADD, ADDS (register) — ADDS, shift or rotate by value
101	0		0000011	ADC, ADCS (register) — ADC, rotate right with extend
101	0		!= 0000011	ADC, ADCS (register) — ADC, shift or rotate by value
101	1		0000011	ADC, ADCS (register) — ADCS, rotate right with extend
101	1		!= 0000011	ADC, ADCS (register) — ADCS, shift or rotate by value
110	0		0000011	SBC, SBCS (register) — SBC, rotate right with extend
110	0		!= 0000011	SBC, SBCS (register) — SBC, shift or rotate by value
110	1		0000011	SBC, SBCS (register) — SBCS, rotate right with extend
110	1		!= 0000011	SBC, SBCS (register) — SBCS, shift or rotate by value
111	0		0000011	RSC, RSCS (register) — RSC, rotate right with extend
111	0		!= 0000011	RSC, RSCS (register) — RSC, shift or rotate by value
111	1		0000011	RSC, RSCS (register) — RSCS, rotate right with extend
111	1		!= 0000011	RSC, RSCS (register) — RSCS, shift or rotate by value

Integer Test and Compare (two register, immediate shift)

The encodings in this section are decoded from [Data-processing register \(immediate shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	opc		1	Rn				(0)	(0)	(0)	(0)	imm5				stype		0	Rm				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details
opc	imm5:type	
00	0000011	TST (register) — Rotate right with extend
00	!= 0000011	TST (register) — Shift or rotate by value
01	0000011	TEQ (register) — Rotate right with extend
01	!= 0000011	TEQ (register) — Shift or rotate by value
10	0000011	CMP (register) — Rotate right with extend
10	!= 0000011	CMP (register) — Shift or rotate by value
11	0000011	CMN (register) — Rotate right with extend
11	!= 0000011	CMN (register) — Shift or rotate by value

Logical Arithmetic (three register, immediate shift)

The encodings in this section are decoded from [Data-processing register \(immediate shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	1	opc		S	Rn				Rd				imm5				stype		0	Rm				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details
opc	S	
00	0	ORR, ORRS (register) — ORR, rotate right with extend
00	0	ORR, ORRS (register) — ORR, shift or rotate by value
00	1	ORR, ORRS (register) — ORRS, rotate right with extend
00	1	ORR, ORRS (register) — ORRS, shift or rotate by value
01	0	MOV, MOVS (register) — MOV, rotate right with extend
01	0	MOV, MOVS (register) — MOV, shift or rotate by value
01	1	MOV, MOVS (register) — MOVS, rotate right with extend
01	1	MOV, MOVS (register) — MOVS, shift or rotate by value
10	0	BIC, BICS (register) — BIC, rotate right with extend
10	0	BIC, BICS (register) — BIC, shift or rotate by value
10	1	BIC, BICS (register) — BICS, rotate right with extend
10	1	BIC, BICS (register) — BICS, shift or rotate by value
11	0	MVN, MVNS (register) — MVN, rotate right with extend
11	0	MVN, MVNS (register) — MVN, shift or rotate by value
11	1	MVN, MVNS (register) — MVNS, rotate right with extend
11	1	MVN, MVNS (register) — MVNS, shift or rotate by value

Data-processing immediate

The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	1	op0						op1																					
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction details
op0	op1	
0x		Integer Data Processing (two register and immediate)
10	00	Move Halfword (immediate)
10	10	Move Special Register and Hints (immediate)
10	x1	Integer Test and Compare (one register and immediate)
11		Logical Arithmetic (two register and immediate)

Integer Data Processing (two register and immediate)

The encodings in this section are decoded from [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	0	opc			S	Rn				Rd				imm12											
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields			Instruction Details
opc	S	Rn	
000	0		AND, ANDS (immediate) — AND
000	1		AND, ANDS (immediate) — ANDS
001	0		EOR, EORS (immediate) — EOR
001	1		EOR, EORS (immediate) — EORS
010	0	1101	SUB, SUBS (SP minus immediate) — SUB
010	0	1111	ADR — A2
010	0	!= 11x1	SUB, SUBS (immediate) — SUB
010	1	1101	SUB, SUBS (SP minus immediate) — SUBS
010	1	!= 1101	SUB, SUBS (immediate) — SUBS
011	0		RSB, RSBS (immediate) — RSB
011	1		RSB, RSBS (immediate) — RSBS
100	0	1101	ADD, ADDS (SP plus immediate) — ADD
100	0	1111	ADR — A1
100	0	!= 11x1	ADD, ADDS (immediate) — ADD
100	1	1101	ADD, ADDS (SP plus immediate) — ADDS
100	1	!= 1101	ADD, ADDS (immediate) — ADDS
101	0		ADC, ADCS (immediate) — ADC
101	1		ADC, ADCS (immediate) — ADCS
110	0		SBC, SBCS (immediate) — SBC
110	1		SBC, SBCS (immediate) — SBCS
111	0		RSC, RSCS (immediate) — RSC
111	1		RSC, RSCS (immediate) — RSCS

Move Halfword (immediate)

The encodings in this section are decoded from [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	H	0	0	imm4				Rd				imm12											
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

Instruction Details

H	
0	MOV, MOVS (immediate)
1	MOVT

Move Special Register and Hints (immediate)

The encodings in this section are decoded from [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	1	1	0	R	1	0				imm4	(1)	(1)	(1)	(1)															
cond																				imm12											

The following constraints also apply to this encoding: cond != '1111'

Decode fields

Instruction Details

Condition

R:imm4	imm12		
00000	xxxxx101xxxx	Reserved hint, behaves as NOP	-
00000	xxxx0x1xxxxx	Reserved hint, behaves as NOP	-
00000	xxxx000x1xxx	Reserved hint, behaves as NOP	-
00000	xxxx00000000	NOP	-
00000	xxxx00000001	YIELD	-
00000	xxxx00000010	WFE	-
00000	xxxx00000011	WFI	-
00000	xxxx00000100	SEV	-
00000	xxxx00000101	SEVL	-
00000	xxxx0000011x	Reserved hint, behaves as NOP	-
00000	xxxx00010xx1	Reserved hint, behaves as NOP	-
00000	xxxx00010000	ESB	FEAT_RAS
00000	xxxx00010010	TSB	FEAT_TRF
00000	xxxx00010100	CSDB	-
00000	xxxx00010110	CLRBHB	FEAT_CLRBHB
00000	xxxx0100xxxx	Reserved hint, behaves as NOP	-
00000	xxxx10xxxxxx	Reserved hint, behaves as NOP	-
00000	xxxx11x0xxxx	Reserved hint, behaves as NOP	-
00000	xxxx111xxxxx	DBG	-
!= 00000		MSR (immediate)	-

Integer Test and Compare (one register and immediate)

The encodings in this section are decoded from [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	opc		1	Rn				(0)	(0)	(0)	(0)	imm12											
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

Instruction Details

opc	
00	TST (immediate)
01	TEQ (immediate)
10	CMP (immediate)
11	CMN (immediate)

Logical Arithmetic (two register and immediate)

The encodings in this section are decoded from [Data-processing immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	1	opc		S	Rn				Rd				imm12											
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details
opc	S	
00	0	ORR, ORRS (immediate) — ORR
00	1	ORR, ORRS (immediate) — ORRS
01	0	MOV, MOVS (immediate) — MOV
01	1	MOV, MOVS (immediate) — MOVS
10	0	BIC, BICS (immediate) — BIC
10	1	BIC, BICS (immediate) — BICS
11	0	MVN, MVNS (immediate) — MVN
11	1	MVN, MVNS (immediate) — MVNS

Load/Store Word, Unsigned Byte (immediate, literal)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0	1	0	P	U	op2	W	op1	Rn				Rt				imm12															
cond																																			

The following constraints also apply to this encoding: cond != '1111'

Decode fields				Instruction Details
P:W	op2	op1	Rn	
00	0	0		STR (immediate) — Post-indexed
00	0	1	!= 1111	LDR (immediate) — Post-indexed
00	1	0		STRB (immediate) — Post-indexed
00	1	1	!= 1111	LDRB (immediate) — Post-indexed
01	0	0		STRT
01	0	1		LDRT
01	1	0		STRBT
01	1	1		LDRBT
10	0	0		STR (immediate) — Offset
10	0	1	!= 1111	LDR (immediate) — Offset
10	1	0		STRB (immediate) — Offset
10	1	1	!= 1111	LDRB (immediate) — Offset
11	0	0		STR (immediate) — Pre-indexed
11	0	1	!= 1111	LDR (immediate) — Pre-indexed
11	1	0		STRB (immediate) — Pre-indexed
11	1	1	!= 1111	LDRB (immediate) — Pre-indexed
!= 01	0	1	1111	LDR (literal)
!= 01	1	1	1111	LDRB (literal)

Load/Store Word, Unsigned Byte (register)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	P	U	op2	W	op1	Rn				Rt				imm5				stype		0	Rm				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields				Instruction Details
P	op2	W	op1	
0	0	0	0	STR (register) — Post-indexed
0	0	0	1	LDR (register) — Post-indexed
0	0	1	0	STRT
0	0	1	1	LDRT
0	1	0	0	STRB (register) — Post-indexed
0	1	0	1	LDRB (register) — Post-indexed
0	1	1	0	STRBT
0	1	1	1	LDRBT
1	0	0	0	STR (register) — Offset
1	0	0	1	LDR (register) — Offset
1	0	1	0	STR (register) — Pre-indexed
1	0	1	1	LDR (register) — Pre-indexed
1	1	0	0	STRB (register) — Offset
1	1	0	1	LDRB (register) — Offset
1	1	1	0	STRB (register) — Pre-indexed
1	1	1	1	LDRB (register) — Pre-indexed

Media instructions

The encodings in this section are decoded from [A32 instruction set encoding](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0	1	1	op0																			op1		1				
cond																																

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction details
op0	op1	
00xxx		Parallel Arithmetic
01000	001	UNALLOCATED
01000	101	SEL
01000	xx0	Pack Halfword
01001	!= x11	UNALLOCATED
0110x	001	UNALLOCATED
0110x	xx0	UNALLOCATED
01x10	001	Saturate 16-bit
01x10	101	UNALLOCATED
01x11	x01	Reverse Bit/Byte
01x1x	xx0	Saturate 32-bit
01xxx	011	Extend and Add
01xxx	111	UNALLOCATED
10xxx		Signed multiply, Divide
11000	000	Unsigned Sum of Absolute Differences
11001	000	UNALLOCATED
1100x	10x	UNALLOCATED
1101x	111	UNALLOCATED
1110x	x00	Bitfield Insert
11110	111	UNALLOCATED
11111	111	Permanently UNDEFINED
11x0x	001	UNALLOCATED
11x0x	x1x	UNALLOCATED
11x1x	011	UNALLOCATED
11x1x	x0x	UNALLOCATED
11x1x	x10	Bitfield Extract
x110x	101	UNALLOCATED

Parallel Arithmetic

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	0	op1			Rn			Rd			(1)	(1)	(1)	(1)	B	op2		1	Rm					
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details	
op1	B	op2	
x00			UNALLOCATED
!= x00	1	01	UNALLOCATED
!= x00	1	10	UNALLOCATED
001	0	00	SADD16
001	0	01	SASX
001	0	10	SSAX
001	0	11	SSUB16
001	1	00	SADD8
001	1	11	SSUB8
010	0	00	QADD16
010	0	01	QASX
010	0	10	QSAX
010	0	11	QSUB16
010	1	00	QADD8
010	1	11	QSUB8
011	0	00	SHADD16
011	0	01	SHASX
011	0	10	SHSAX
011	0	11	SHSUB16
011	1	00	SHADD8
011	1	11	SHSUB8
101	0	00	UADD16
101	0	01	UASX
101	0	10	USAX
101	0	11	USUB16
101	1	00	UADD8
101	1	11	USUB8
110	0	00	UQADD16
110	0	01	UQASX
110	0	10	UQSAX
110	0	11	UQSUB16
110	1	00	UQADD8
110	1	11	UQSUB8
111	0	00	UHADD16
111	0	01	UHASX
111	0	10	UHSAX
111	0	11	UHSUB16
111	1	00	UHADD8
111	1	11	UHSUB8

Pack Halfword

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	0	0	Rn				Rd				imm5				tb	0	1	Rm				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

tb

Instruction Details

0	PKHBT, PKHTB — PKHBT
1	PKHBT, PKHTB — PKHTB

Saturate 16-bit

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	U	1	0	sat imm				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rn			
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

U

Instruction Details

0	SSAT16
1	USAT16

Reverse Bit/Byte

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	o1	1	1	(1)	(1)	(1)	(1)	Rd			(1)			(1)	(1)	(1)	o2	0	1	1	Rm		
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

o1

o2

Instruction Details

0	0	REV
0	1	REV16
1	0	RBIT
1	1	REVSH

Saturate 32-bit

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	U	1	sat imm				Rd				imm5				sh	0	1	Rn					
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

U

sh

Instruction Details

0	0	SSAT — Logical shift left
0	1	SSAT — Arithmetic shift right
1	0	USAT — Logical shift left
1	1	USAT — Arithmetic shift right

Extend and Add

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	U	op			Rn			Rd			rotate			(0)	(0)	0	1	1	1	Rm			
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields																Instruction Details															
U				op				Rn																							
				01												UNALLOCATED															
0				00				1111								SXTB16															
0				00				!= 1111								SXTAB16															
0				10				1111								SXTB															
0				10				!= 1111								SXTAB															
0				11				1111								SXTH															
0				11				!= 1111								SXTAH															
1				00				1111								UXTB16															
1				00				!= 1111								UXTAB16															
1				10				1111								UXTB															
1				10				!= 1111								UXTAB															
1				11				1111								UXTH															
1				11				!= 1111								UXTAH															

Signed multiply, Divide

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	0	op1			Rd				Ra				Rm				op2		1	Rn				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details	
op1	Ra	op2	
0xx		1xx	UNALLOCATED
0x1		0x1	UNALLOCATED
0x1		010	UNALLOCATED
000	1111	000	SMUAD, SMUADX — SMUAD
000	1111	001	SMUAD, SMUADX — SMUADX
000	1111	010	SMUSD, SMUSDX — SMUSD
000	1111	011	SMUSD, SMUSDX — SMUSDX
000	!= 1111	000	SMLAD, SMLADX — SMLAD
000	!= 1111	001	SMLAD, SMLADX — SMLADX
000	!= 1111	010	SMLSD, SMLSDX — SMLSD
000	!= 1111	011	SMLSD, SMLSDX — SMLSDX
001		000	SDIV
010		0xx	UNALLOCATED
011		000	UDIV
100		000	SMLALD, SMLALDX — SMLALD
100		001	SMLALD, SMLALDX — SMLALDX
100		010	SMLS LD, SMLS LDX — SMLS LD
100		011	SMLS LD, SMLS LDX — SMLS LDX
100		1xx	UNALLOCATED
101		01x	UNALLOCATED
101		10x	UNALLOCATED
101		110	SMMLS, SMMLSR — SMMLS
101		111	SMMLS, SMMLSR — SMMLSR
101	1111	000	SMMUL, SMMULR — SMMUL
101	1111	001	SMMUL, SMMULR — SMMULR
101	!= 1111	000	SMMLA, SMMLAR — SMMLA
101	!= 1111	001	SMMLA, SMMLAR — SMMLAR
11x			UNALLOCATED

Unsigned Sum of Absolute Differences

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	0	0	0	Rd				Ra				Rm				0	0	0	1	Rn			
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details	
Ra			
1111		USAD8	
!= 1111		USADA8	

Bitfield Insert

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	1	0	msb				Rd				lsb				0 0 1			Rn					
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields	Instruction Details
Rn	
1111	BFC
!= 1111	BFI

Permanently UNDEFINED

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
!= 1111				0	1	1	1	1	1	1	1	imm12											1	1	1	1	imm4								
cond																																imm4			

The following constraints also apply to this encoding: cond != '1111'

Decode fields	Instruction Details
cond	
1110	UDF
!= 111x	UNALLOCATED

Bitfield Extract

The encodings in this section are decoded from [Media instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	1	1	1	1	1	U	1																							
cond								widthm1				Rd				lsb								1 0 1				Rn			

The following constraints also apply to this encoding: cond != '1111'

Decode fields	Instruction Details
U	
0	SBFX
1	UBFX

Branch, branch with link, and block data transfer

The encodings in this section are decoded from [A32 instruction set encoding](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	op0																									

Decode fields	Instruction details	
cond	op0	
1111	0	Exception Save/Restore
	1	Branch (immediate)
!= 1111	0	Load/Store Multiple

Exception Save/Restore

The encodings in this section are decoded from [Branch, branch with link, and block data transfer](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	S	W	L																				
								Rn								op								mode							

Decode fields				Instruction Details
P	U	S	L	
		0	0	UNALLOCATED
		1	1	UNALLOCATED
0	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Decrement After
0	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Decrement After
0	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Increment After
0	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Increment After
1	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Decrement Before
1	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Decrement Before
1	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Increment Before
1	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Increment Before

Branch (immediate)

The encodings in this section are decoded from [Branch, branch with link, and block data transfer](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	H	imm24																							

Decode fields		H	Instruction Details
cond			
1 1 1 1			BL, BLX (immediate) — A2
!= 1 1 1 1		0	B
!= 1 1 1 1		1	BL, BLX (immediate) — A1

Load/Store Multiple

The encodings in this section are decoded from [Branch, branch with link, and block data transfer](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	0	0	P	U	op	W	L	Rn				register_list															
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields				register_list	Instruction Details
P	U	op	L		
		1	0		STM (User registers)
		1	1	0xxxxxxxxxxxxxxxxxxx	LDM (User registers)
		1	1	1xxxxxxxxxxxxxxxxxxx	LDM (exception return)
0	0	0	0		STMDA, STMED
0	0	0	1		LDMDA, LDMFA
0	1	0	0		STM, STMIA, STMEA
0	1	0	1		LDM, LDMIA, LDMFD
1	0	0	0		STMDB, STMFD
1	0	0	1		LDMDB, LDMEA
1	1	0	0		STMIB, STMFA
1	1	0	1		LDMIB, LDMED

System register access, Advanced SIMD, floating-point, and Supervisor call

The encodings in this section are decoded from [A32 instruction set encoding](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	op0															op1							op2			

Decode fields				Instruction details
cond	op0	op1	op2	
1111	!= 11	1x		Unconditional Advanced SIMD and floating-point instructions
	11			Supervisor call
	!= 11	0x		UNALLOCATED
!= 1111	0x	1x		Advanced SIMD and System register load/store and 64-bit move
!= 1111	10	10	0	Floating-point data-processing
!= 1111	10	11	0	UNALLOCATED
!= 1111	10	1x	1	Advanced SIMD and System register 32-bit move

Unconditional Advanced SIMD and floating-point instructions

The encodings in this section are decoded from [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	!= 11x				op1											1	op2	op3			op4		op5				
							op0																									

The following constraints also apply to this encoding: !(op0 IN {'11x'})

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0xx			0x			Advanced SIMD three registers of the same length extension
0xx			1x			UNALLOCATED
100		0	!= 00	0	0	Floating-point conditional select
100		0	!= 00	1	0	UNALLOCATED
101	00xxxx	0	!= 00		0	Floating-point minNum/maxNum
101	01xxxx	0	!= 00		0	UNALLOCATED
101	10xxxx	0	!= 00		0	UNALLOCATED
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	110001	0	!= 00	1	0	UNALLOCATED
101	11001x	0	!= 00	1	0	UNALLOCATED
101	1101xx	0	!= 00	1	0	UNALLOCATED
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
101	11xxxx	0	!= 00	0	0	UNALLOCATED
10x		0	00			Advanced SIMD and floating-point multiply with accumulate
10x		0	!= 00		1	UNALLOCATED
10x		1	0x			Advanced SIMD and floating-point dot product
10x		1	1x			UNALLOCATED

Advanced SIMD three registers of the same length extension

The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	op1	D	op2		Vn		Vd			1	op3	0	op4	N	Q	M	U							Vm	

Decode fields						Instruction Details	Condition
op1	op2	op3	op4	Q	U		
		0	1		0	UNALLOCATED	-
	1x	0	0	0	0	VCMLA — 64-bit SIMD vector	FEAT_FCMA
	1x	0	0	1	0	VCMLA — 128-bit SIMD vector	FEAT_FCMA
x0	0x	0	0		0	UNALLOCATED	-
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector	FEAT_FCMA
0x	x1	1			0	UNALLOCATED	-
0x	0x				1	UNALLOCATED	-
0x	1x	0	1		1	UNALLOCATED	-
0x	10	1	0	0	0	UNALLOCATED	-
0x	11	1			1	UNALLOCATED	-
00	00	1	0	0	0	UNALLOCATED	-
00	00	1	0	1	0	VMMLA	FEAT_AA32BF16
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector	FEAT_AA32BF16
00	10	0	0	0	1	VFMAL (vector) — 64-bit SIMD vector	FEAT_FHM
00	10	0	0	1	1	VFMAL (vector) — 128-bit SIMD vector	FEAT_FHM
00	10	1	0	0	1	UNALLOCATED	-
00	10	1	0	1	0	VSMMLA	FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA	FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	11	0	0		1	VFMAb, VFMAb (BFloat16, vector)	FEAT_AA32BF16
01	00	1			0	UNALLOCATED	-
01	10	0	0	0	1	VFMSL (vector) — 64-bit SIMD vector	FEAT_FHM
01	10	0	0	1	1	VFMSL (vector) — 128-bit SIMD vector	FEAT_FHM
01	10	1			1	UNALLOCATED	-
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector	FEAT_AA32I8MM
01	11	0	0		1	UNALLOCATED	-
1x		0			1	UNALLOCATED	-
1x		1				UNALLOCATED	-

Floating-point conditional select

The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	cc	Vn				Vd				1	0	!= 00	N	0	M	0	Vm					
size																															

size

The following constraints also apply to this encoding: size != '00'

Decode fields		Instruction Details	Condition
cc	size		
00	01	VSELEQ, VSELGE, VSELGT, VSELVS — Equal, half-precision scalar	FEAT_FP16
00	10	VSELEQ, VSELGE, VSELGT, VSELVS — Equal, single-precision scalar	-
00	11	VSELEQ, VSELGE, VSELGT, VSELVS — Equal, double-precision scalar	-
01	01	VSELEQ, VSELGE, VSELGT, VSELVS — Unordered, half-precision scalar	FEAT_FP16
01	10	VSELEQ, VSELGE, VSELGT, VSELVS — Unordered, single-precision scalar	-
01	11	VSELEQ, VSELGE, VSELGT, VSELVS — Unordered, double-precision scalar	-
10	01	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than or Equal, half-precision scalar	FEAT_FP16
10	10	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than or Equal, single-precision scalar	-
10	11	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than or Equal, double-precision scalar	-
11	01	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than, half-precision scalar	FEAT_FP16
11	10	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than, single-precision scalar	-
11	11	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than, double-precision scalar	-

Floating-point minNum/maxNum

The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn			Vd			1			0	!= 00	N	op	M	0	Vm				
																	size														

The following constraints also apply to this encoding: size != '00'

Decode fields		Instruction Details	Condition
size	op		
01	0	VMAXNM — Half-precision scalar	FEAT_FP16
01	1	VMINNM — Half-precision scalar	FEAT_FP16
10	0	VMAXNM — Single-precision scalar	-
10	1	VMINNM — Single-precision scalar	-
11	0	VMAXNM — Double-precision scalar	-
11	1	VMINNM — Double-precision scalar	-

Floating-point extraction and insertion

The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	!= 00	op	1	M	0	Vm				
																	size														

The following constraints also apply to this encoding: size != '00'

Decode fields		Instruction Details	Condition
size	op		
x1		UNALLOCATED	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16

Floating-point directed convert to integer

The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	op1	RM	Vd			1	0	!= 00	op	1	M	0	Vm						
																	size														

The following constraints also apply to this encoding: size != '00'

Decode fields				Instruction Details	Condition
op1	RM	size	op		
0		!= 00	1	UNALLOCATED	-
0	00	01	0	VRINTA (floating-point) — Half-precision scalar	FEAT_FP16
0	00	10	0	VRINTA (floating-point) — Single-precision scalar	-
0	00	11	0	VRINTA (floating-point) — Double-precision scalar	-
0	01	01	0	VRINTN (floating-point) — Half-precision scalar	FEAT_FP16
0	01	10	0	VRINTN (floating-point) — Single-precision scalar	-
0	01	11	0	VRINTN (floating-point) — Double-precision scalar	-
0	10	01	0	VRINTP (floating-point) — Half-precision scalar	FEAT_FP16
0	10	10	0	VRINTP (floating-point) — Single-precision scalar	-
0	10	11	0	VRINTP (floating-point) — Double-precision scalar	-
0	11	01	0	VRINTM (floating-point) — Half-precision scalar	FEAT_FP16
0	11	10	0	VRINTM (floating-point) — Single-precision scalar	-
0	11	11	0	VRINTM (floating-point) — Double-precision scalar	-
1	00	01		VCVTA (floating-point) — Half-precision scalar	FEAT_FP16
1	00	10		VCVTA (floating-point) — Single-precision scalar	-
1	00	11		VCVTA (floating-point) — Double-precision scalar	-
1	01	01		VCVTN (floating-point) — Half-precision scalar	FEAT_FP16
1	01	10		VCVTN (floating-point) — Single-precision scalar	-
1	01	11		VCVTN (floating-point) — Double-precision scalar	-
1	10	01		VCVTP (floating-point) — Half-precision scalar	FEAT_FP16
1	10	10		VCVTP (floating-point) — Single-precision scalar	-
1	10	11		VCVTP (floating-point) — Double-precision scalar	-
1	11	01		VCVTM (floating-point) — Half-precision scalar	FEAT_FP16
1	11	10		VCVTM (floating-point) — Single-precision scalar	-
1	11	11		VCVTM (floating-point) — Double-precision scalar	-

Advanced SIMD and floating-point multiply with accumulate

The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D	op2	Vn				Vd				1	0	0	0	N	Q	M	U	Vm				

Decode fields				Instruction Details	Condition
op1	op2	Q	U		
0		0	0	VCMLA (by element) — 64-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0		1	0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0	00	0	1	VFMAL (by scalar) — 64-bit SIMD vector	FEAT_FHM
0	00	1	1	VFMAL (by scalar) — 128-bit SIMD vector	FEAT_FHM
0	01	0	1	VFMSL (by scalar) — 64-bit SIMD vector	FEAT_FHM
0	01	1	1	VFMSL (by scalar) — 128-bit SIMD vector	FEAT_FHM
0	10		1	UNALLOCATED	-
0	11		1	VFMAb, VFMAbT (BFloat16, by scalar)	FEAT_AA32BF16
1			1	UNALLOCATED	-
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point	FEAT_FCMA
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point	FEAT_FCMA

Advanced SIMD and floating-point dot product

The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D	op2			Vn				Vd			1	1	0	op4	N	O	M	U			Vm	

Decode fields					Instruction Details	Condition
op1	op2	op4	Q	U		
	×0	0			UNALLOCATED	-
	×1				UNALLOCATED	-
0	00	1		1	UNALLOCATED	-
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector	FEAT_AA32BF16
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	FEAT_AA32BF16
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	10	1			UNALLOCATED	-

Supervisor call

The encodings in this section are decoded from [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				cond	1	1	1	1																							

Decode fields	Instruction details
cond	
1111	UNALLOCATED
!= 1111	SVC

Advanced SIMD and System register load/store and 64-bit move

The encodings in this section are decoded from [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				cond	1	1	0		op0											1		op1									

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction details
op0	op1	
00×0	0×	Advanced SIMD and floating-point 64-bit move
00×0	11	System register 64-bit move
	10	UNALLOCATED
!= 00×0	0×	Advanced SIMD and floating-point load/store
!= 00×0	11	System register load/store

Advanced SIMD and floating-point 64-bit move

The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	0	0	D	0	op	Rt2				Rt				1	0	size	opc2	M	o3	Vm					
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields					Instruction Details
D	op	size	opc2	o3	
0					UNALLOCATED
1		0x			UNALLOCATED
1		1x	00	0	UNALLOCATED
1		1x	!= 00		UNALLOCATED
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — From general-purpose registers
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — From general-purpose registers
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — To general-purpose registers
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — To general-purpose registers

System register 64-bit move

The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	0	0	D	0	L	Rt2				Rt				1	1	1	cp15	opc1				CRm			
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details
D	L	
0		UNALLOCATED
1	0	MCRR
1	1	MRRC

Advanced SIMD and floating-point load/store

The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	P	U	D	W	L	Rn				Vd				1	0	size	imm8								
cond																															

The following constraints also apply to this encoding: cond != '1111' && !(P == '0' && U == '0' && W == '0')

Decode fields						Instruction Details		Condition
P	U	W	L	Rn	size	imm8		
		1			0x		UNALLOCATED	-
0	0	1			1x		UNALLOCATED	-
0	1		0		10		VSTM, VSTMDB, VSTMIA — A2 Increment After	-
0	1		0		11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA — A1 Increment After	-
0	1		0		11	xxxxxxxx1	FSTMDBX, FSTMIAX — Increment After	-
0	1		1		10		VLDM, VLDMDB, VLDMIA — A2 Increment After	-
0	1		1		11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA — A1 Increment After	-
0	1		1		11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Increment After	-
0	1	0			0x		UNALLOCATED	-
1		0			00		UNALLOCATED	-
1		0	0		01		VSTR — Half-precision scalar	FEAT_FP16
1		0	0		10		VSTR — Single-precision scalar	-
1		0	0		11		VSTR — Double-precision scalar	-
1		0	1	1111	01		VLDR (literal) — Half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — Single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — Double-precision scalar	-
1		0	1	!= 1111	01		VLDR (immediate) — Half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — Single-precision scalar	-
1		0	1	!= 1111	11		VLDR (immediate) — Double-precision scalar	-
1	0	1	0		10		VSTM, VSTMDB, VSTMIA — A2 Decrement Before	-
1	0	1	0		11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA — A1 Decrement Before	-
1	0	1	0		11	xxxxxxxx1	FSTMDBX, FSTMIAX — Decrement Before	-
1	0	1	1		10		VLDM, VLDMDB, VLDMIA — A2 Decrement Before	-
1	0	1	1		11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA — A1 Decrement Before	-
1	0	1	1		11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before	-
1	1	1			1x		UNALLOCATED	-

System register load/store

The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	P	U	D	W	L	Rn				CRd				1	1	1	cp15	imm8							
cond																															

The following constraints also apply to this encoding: cond != '1111' && !(P == '0' && U == '0' && W == '0')

Decode fields						Instruction Details
P:U:W	D	L	Rn	CRd	cp15	
0x1	0	0		0101	0	STC — Post-indexed
0x1	0	1	!= 1111	0101	0	LDC (immediate) — Post-indexed
010	0	0		0101	0	STC — Unindexed
010	0	1	!= 1111	0101	0	LDC (immediate) — Unindexed
1x0	0	0		0101	0	STC — Offset
1x0	0	1	!= 1111	0101	0	LDC (immediate) — Offset
1x1	0	0		0101	0	STC — Pre-indexed
1x1	0	1	!= 1111	0101	0	LDC (immediate) — Pre-indexed
!= 000	0			0101	1	UNALLOCATED
!= 000	0			!= 0101		UNALLOCATED
!= 000	0	1	1111	0101	0	LDC (literal)
!= 000	1					UNALLOCATED

Floating-point data-processing

The encodings in this section are decoded from [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	1	1	1	0					op0												1	0				op1		0			
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction details
op0	op1	
1x11	0	Floating-point move immediate
1x11	1	Floating-point data-processing (two registers)
!= 1x11		Floating-point data-processing (three registers)

Floating-point move immediate

The encodings in this section are decoded from [Floating-point data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	1	1	1	0	1	D	1	1				imm4H				Vd				1	0	size	(0)	0	(0)	0			imm4L		
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields	Instruction Details		Condition
size			
00	UNALLOCATED		-
01	VMOV (immediate) — Half-precision scalar		FEAT_FP16
10	VMOV (immediate) — Single-precision scalar		-
11	VMOV (immediate) — Double-precision scalar		-

Floating-point data-processing (two registers)

The encodings in this section are decoded from [Floating-point data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	1	1	1	0	1	D	1	1	o1		opc2					Vd				1	0	size	o3	1	M	0			Vm		
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields				Instruction Details	Condition
o1	opc2	size	o3		
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000	01	1	VABS — Half-precision scalar	FEAT_FP16
0	000	10	0	VMOV (register) — Single-precision scalar	-
0	000	10	1	VABS — Single-precision scalar	-
0	000	11	0	VMOV (register) — Double-precision scalar	-
0	000	11	1	VABS — Double-precision scalar	-
0	001	01	0	VNEG — Half-precision scalar	FEAT_FP16
0	001	01	1	VSQRT — Half-precision scalar	FEAT_FP16
0	001	10	0	VNEG — Single-precision scalar	-
0	001	10	1	VSQRT — Single-precision scalar	-
0	001	11	0	VNEG — Double-precision scalar	-
0	001	11	1	VSQRT — Double-precision scalar	-
0	010	01		UNALLOCATED	-
0	010	10	0	VCVTB — Half-precision to single-precision	-
0	010	10	1	VCVTT — Half-precision to single-precision	-
0	010	11	0	VCVTB — Half-precision to double-precision	-
0	010	11	1	VCVTT — Half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB — Single-precision to half-precision	-
0	011	10	1	VCVTT — Single-precision to half-precision	-
0	011	11	0	VCVTB — Double-precision to half-precision	-
0	011	11	1	VCVTT — Double-precision to half-precision	-
0	100	01	0	VCMP — A1 Half-precision scalar	FEAT_FP16
0	100	01	1	VCMP — A1 Half-precision scalar	FEAT_FP16
0	100	10	0	VCMP — A1 Single-precision scalar	-
0	100	10	1	VCMP — A1 Single-precision scalar	-
0	100	11	0	VCMP — A1 Double-precision scalar	-
0	100	11	1	VCMP — A1 Double-precision scalar	-
0	101	01	0	VCMP — A2 Half-precision scalar	FEAT_FP16
0	101	01	1	VCMP — A2 Half-precision scalar	FEAT_FP16
0	101	10	0	VCMP — A2 Single-precision scalar	-
0	101	10	1	VCMP — A2 Single-precision scalar	-
0	101	11	0	VCMP — A2 Double-precision scalar	-
0	101	11	1	VCMP — A2 Double-precision scalar	-
0	110	01	0	VRINTR — Half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — Half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — Single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — Single-precision scalar	-
0	110	11	0	VRINTR — Double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — Double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — Half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — Single-precision scalar	-

Decode fields				Instruction Details	Condition
o1	opc2	size	o3		
0	111	10	1	VCVT (between double-precision and single-precision) — Single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — Double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — Double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — Half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — Single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — Double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point) — Half-precision scalar	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point) — Single-precision scalar	-
1	01x	11		VCVT (between floating-point and fixed-point, floating-point) — Double-precision scalar	-
1	100	01	0	VCVTR — Half-precision scalar	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point) — Half-precision scalar	FEAT_FP16
1	100	10	0	VCVTR — Single-precision scalar	-
1	100	10	1	VCVT (floating-point to integer, floating-point) — Single-precision scalar	-
1	100	11	0	VCVTR — Double-precision scalar	-
1	100	11	1	VCVT (floating-point to integer, floating-point) — Double-precision scalar	-
1	101	01	0	VCVTR — Half-precision scalar	FEAT_FP16
1	101	01	1	VCVT (floating-point to integer, floating-point) — Half-precision scalar	FEAT_FP16
1	101	10	0	VCVTR — Single-precision scalar	-
1	101	10	1	VCVT (floating-point to integer, floating-point) — Single-precision scalar	-
1	101	11	0	VCVTR — Double-precision scalar	-
1	101	11	1	VCVT (floating-point to integer, floating-point) — Double-precision scalar	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point) — Half-precision scalar	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point) — Single-precision scalar	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point) — Double-precision scalar	-

Floating-point data-processing (three registers)

The encodings in this section are decoded from [Floating-point data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	o0	D	o1			Vn			Vd			1 0		size	N	o2	M	0	Vm					
cond																															

The following constraints also apply to this encoding: cond != '1111' && !(o0 == '1' && o1 == '11')

Decode fields				Instruction Details	Condition
o0	o1	size	o2		
0		00		UNALLOCATED	-
0	00	01	0	VMLA (floating-point) — Half-precision scalar	FEAT_FP16
0	00	01	1	VMLS (floating-point) — Half-precision scalar	FEAT_FP16
0	00	10	0	VMLA (floating-point) — Single-precision scalar	-
0	00	10	1	VMLS (floating-point) — Single-precision scalar	-
0	00	11	0	VMLA (floating-point) — Double-precision scalar	-
0	00	11	1	VMLS (floating-point) — Double-precision scalar	-
0	01	01	0	VNMLS — Half-precision scalar	FEAT_FP16
0	01	01	1	VNMLA — Half-precision scalar	FEAT_FP16
0	01	10	0	VNMLS — Single-precision scalar	-
0	01	10	1	VNMLA — Single-precision scalar	-
0	01	11	0	VNMLS — Double-precision scalar	-
0	01	11	1	VNMLA — Double-precision scalar	-
0	10	01	0	VMUL (floating-point) — Half-precision scalar	FEAT_FP16
0	10	01	1	VNMUL — Half-precision scalar	FEAT_FP16
0	10	10	0	VMUL (floating-point) — Single-precision scalar	-
0	10	10	1	VNMUL — Single-precision scalar	-
0	10	11	0	VMUL (floating-point) — Double-precision scalar	-
0	10	11	1	VNMUL — Double-precision scalar	-
0	11	01	0	VADD (floating-point) — Half-precision scalar	FEAT_FP16
0	11	01	1	VSUB (floating-point) — Half-precision scalar	FEAT_FP16
0	11	10	0	VADD (floating-point) — Single-precision scalar	-
0	11	10	1	VSUB (floating-point) — Single-precision scalar	-
0	11	11	0	VADD (floating-point) — Double-precision scalar	-
0	11	11	1	VSUB (floating-point) — Double-precision scalar	-
1	00	01	0	VDIV — Half-precision scalar	FEAT_FP16
1	00	10	0	VDIV — Single-precision scalar	-
1	00	11	0	VDIV — Double-precision scalar	-
1	00	!= 00	1	UNALLOCATED	-
1	01	01	0	VFNMS — Half-precision scalar	FEAT_FP16
1	01	01	1	VFNMA — Half-precision scalar	FEAT_FP16
1	01	10	0	VFNMS — Single-precision scalar	-
1	01	10	1	VFNMA — Single-precision scalar	-
1	01	11	0	VFNMS — Double-precision scalar	-
1	01	11	1	VFNMA — Double-precision scalar	-
1	10	01	0	VFMA — Half-precision scalar	FEAT_FP16
1	10	01	1	VFMS — Half-precision scalar	FEAT_FP16
1	10	10	0	VFMA — Single-precision scalar	-
1	10	10	1	VFMS — Single-precision scalar	-
1	10	11	0	VFMA — Double-precision scalar	-
1	10	11	1	VFMS — Double-precision scalar	-
1	!= 11	00		UNALLOCATED	-

Advanced SIMD and System register 32-bit move

The encodings in this section are decoded from [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	op0												1	op1						1				
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction details
op0	op1	
000	000	UNALLOCATED
000	001	Floating-point 16-bit move
000	010	Floating-point 32-bit move
000	10x	UNALLOCATED
0x1	010	UNALLOCATED
10x	010	UNALLOCATED
111	010	Floating-point move special register
x10	010	UNALLOCATED
	011	Advanced SIMD 8/16/32-bit element move/duplicate
	11x	System register 32-bit move
!= 000	x0x	UNALLOCATED

Floating-point 16-bit move

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	0	0	op	Vn				Rt			1	0	0	1	N	(0)	(0)	1	(0)	(0)	(0)	(0)	
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details	Condition
op			
0	VMOV (between general-purpose register and half-precision) — From general-purpose register		FEAT_FP16
1	VMOV (between general-purpose register and half-precision) — To general-purpose register		FEAT_FP16

Floating-point 32-bit move

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	0	0	op	Vn				Rt				1	0	1	0	N	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields		Instruction Details
op		
0	VMOV (between general-purpose register and single-precision) — From general-purpose register	
1	VMOV (between general-purpose register and single-precision) — To general-purpose register	

Floating-point move special register

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	1	1	L	reg				Rt			1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)	
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

L

Instruction Details

0	VMSR
1	VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	1	1	1	0	opc1	L	Vn	Rt	1	0	1	1	N	opc2	1	(0)	(0)	(0)	(0)												
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

opc1

L

opc2

Instruction Details

	1		VMOV (scalar to general-purpose register)
0xx	0		VMOV (general-purpose register to scalar)
1xx	0	1x	UNALLOCATED
1x0	0	0x	VDUP (general-purpose register) — 64-bit SIMD vector
1x1	0	0x	VDUP (general-purpose register) — 128-bit SIMD vector

System register 32-bit move

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	1	1	1	0	opc1	L	CRn	Rt	1	1	1	cp15	opc2	1	CRm																
cond																															

The following constraints also apply to this encoding: cond != '1111'

Decode fields

L

Instruction Details

0	MCR
1	MRC

Unconditional instructions

The encodings in this section are decoded from [A32 instruction set encoding](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	op0						op1																				

Decode fields

op0

op1

Instruction details

00x		Miscellaneous
01x		Advanced SIMD data-processing
100	0	Advanced SIMD element or structure load/store
101	0	UNALLOCATED
11x	0	UNALLOCATED
1xx	1	Memory hints and barriers

Miscellaneous

The encodings in this section are decoded from [Unconditional instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	op0																		op1						

Decode fields		Instruction details		Condition
op0	op1			
10000	xx0x	Change Process State		-
10001	0000	SETPAN		FEAT_PAN
10001	0001	UNALLOCATED		-
10001	010x	UNALLOCATED		-
10001	1x0x	UNALLOCATED		-
1000x	xx1x	UNALLOCATED		-
10010	0011	UNALLOCATED		-
10010	0110	UNALLOCATED		-
10010	0111	CONSTRAINED UNPREDICTABLE		-
10010	1x1x	UNALLOCATED		-
10011	0011	UNALLOCATED		-
10011	011x	UNALLOCATED		-
10011	1x1x	UNALLOCATED		-
1001x	0010	UNALLOCATED		-
1001x	xx0x	UNALLOCATED		-
!= 100xx		UNALLOCATED		-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

Change Process State

The encodings in this section are decoded from [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	imod	M	op	(0)	(0)	(0)	(0)	(0)	(0)	(0)	E	A	I	F	0	mode				

Decode fields					Instruction Details	
imod	M	op	I	F	mode	
		1	0	0	0xxxx	SETEND
		1	0	0	1xxxx	UNALLOCATED
		1	0	1		UNALLOCATED
		1	1			UNALLOCATED
00	0	0				UNALLOCATED
00	1	0				CPS, CPSID, CPSIE — Change mode
01		0				UNALLOCATED
10	0	0				CPS, CPSID, CPSIE — Interrupt enable
10	1	0				CPS, CPSID, CPSIE — Interrupt enable and change mode
11	0	0				CPS, CPSID, CPSIE — Interrupt disable
11	1	0				CPS, CPSID, CPSIE — Interrupt disable and change mode

Advanced SIMD data-processing

The encodings in this section are decoded from [Unconditional instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1		op0																	op1						

Decode fields		Instruction details
op0	op1	
0		Advanced SIMD three registers of the same length
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

Advanced SIMD three registers of the same length

The encodings in this section are decoded from [Advanced SIMD data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn				Vd				opc				N	Q	M	o1	Vm				

U	Decode fields				Instruction Details	Condition
	size	opc	Q	o1		
		0000	0	0	VHADD — 64-bit SIMD vector	-
		0000	0	1	VQADD — 64-bit SIMD vector	-
		0000	1	0	VHADD — 128-bit SIMD vector	-
		0000	1	1	VQADD — 128-bit SIMD vector	-
		0001	0	0	VRHADD — 64-bit SIMD vector	-
		0001	1	0	VRHADD — 128-bit SIMD vector	-
		0010	0	0	VHSUB — 64-bit SIMD vector	-
		0010	0	1	VQSUB — 64-bit SIMD vector	-
		0010	1	0	VHSUB — 128-bit SIMD vector	-
		0010	1	1	VQSUB — 128-bit SIMD vector	-
		0011	0	0	VCGT (register) — A1 64-bit SIMD vector	-
		0011	0	1	VCGE (register) — A1 64-bit SIMD vector	-
		0011	1	0	VCGT (register) — A1 128-bit SIMD vector	-
		0011	1	1	VCGE (register) — A1 128-bit SIMD vector	-
		0100	0	0	VSHL (register) — 64-bit SIMD vector	-
		0100	0	1	VQSHL (register) — 64-bit SIMD vector	-
		0100	1	0	VSHL (register) — 128-bit SIMD vector	-
		0100	1	1	VQSHL (register) — 128-bit SIMD vector	-
		0101	0	0	VRSHL — 64-bit SIMD vector	-
		0101	0	1	VQRSHL — 64-bit SIMD vector	-
		0101	1	0	VRSHL — 128-bit SIMD vector	-
		0101	1	1	VQRSHL — 128-bit SIMD vector	-
		0110	0	0	VMAX (integer) — 64-bit SIMD vector	-
		0110	0	1	VMIN (integer) — 64-bit SIMD vector	-
		0110	1	0	VMAX (integer) — 128-bit SIMD vector	-
		0110	1	1	VMIN (integer) — 128-bit SIMD vector	-
		0111	0	0	VABD (integer) — 64-bit SIMD vector	-
		0111	0	1	VABA — 64-bit SIMD vector	-
		0111	1	0	VABD (integer) — 128-bit SIMD vector	-
		0111	1	1	VABA — 128-bit SIMD vector	-
		1001	0	1	VMUL (integer and polynomial) — 64-bit SIMD vector	-
		1001	1	1	VMUL (integer and polynomial) — 128-bit SIMD vector	-
		1010	0	0	VPMAX (integer)	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
0		1000	0	0	VADD (integer) — 64-bit SIMD vector	-
0		1000	0	1	VTST — 64-bit SIMD vector	-
0		1000	1	0	VADD (integer) — 128-bit SIMD vector	-
0		1000	1	1	VTST — 128-bit SIMD vector	-
0		1001	0	0	VMLA (integer) — 64-bit SIMD vector	-
0		1001	1	0	VMLA (integer) — 128-bit SIMD vector	-
0		1011		1	VPADD (integer)	-
0		1011	0	0	VQDMULH — 64-bit SIMD vector	-
0		1011	1	0	VQDMULH — 128-bit SIMD vector	-
0	0x	1100	0	1	VFMA — 64-bit SIMD vector	-

Decode fields					Instruction Details	Condition
U	size	opc	Q	o1		
0	0x	1100	1	1	VFMA — 128-bit SIMD vector	-
0	0x	1101	0	0	VADD (floating-point) — 64-bit SIMD vector	-
0	0x	1101	0	1	VMLA (floating-point) — 64-bit SIMD vector	-
0	0x	1101	1	0	VADD (floating-point) — 128-bit SIMD vector	-
0	0x	1101	1	1	VMLA (floating-point) — 128-bit SIMD vector	-
0	0x	1110		1	UNALLOCATED	-
0	0x	1110	0	0	VCEQ (register) — A2 64-bit SIMD vector	-
0	0x	1110	1	0	VCEQ (register) — A2 128-bit SIMD vector	-
0	0x	1111	0	0	VMAX (floating-point) — 64-bit SIMD vector	-
0	0x	1111	0	1	VRECPS — 64-bit SIMD vector	-
0	0x	1111	1	0	VMAX (floating-point) — 128-bit SIMD vector	-
0	0x	1111	1	1	VRECPS — 128-bit SIMD vector	-
0	00	0001	0	1	VAND (register) — 64-bit SIMD vector	-
0	00	0001	1	1	VAND (register) — 128-bit SIMD vector	-
0	00	1100		0	SHA1C	FEAT_SHA1
0	01	0001	0	1	VBIC (register) — 64-bit SIMD vector	-
0	01	0001	1	1	VBIC (register) — 128-bit SIMD vector	-
0	01	1100		0	SHA1P	FEAT_SHA1
0	1x	1100	0	1	VFMS — 64-bit SIMD vector	-
0	1x	1100	1	1	VFMS — 128-bit SIMD vector	-
0	1x	1101	0	0	VSUB (floating-point) — 64-bit SIMD vector	-
0	1x	1101	0	1	VMLS (floating-point) — 64-bit SIMD vector	-
0	1x	1101	1	0	VSUB (floating-point) — 128-bit SIMD vector	-
0	1x	1101	1	1	VMLS (floating-point) — 128-bit SIMD vector	-
0	1x	1110			UNALLOCATED	-
0	1x	1111	0	0	VMIN (floating-point) — 64-bit SIMD vector	-
0	1x	1111	0	1	VRSQRTS — 64-bit SIMD vector	-
0	1x	1111	1	0	VMIN (floating-point) — 128-bit SIMD vector	-
0	1x	1111	1	1	VRSQRTS — 128-bit SIMD vector	-
0	10	0001	0	1	VORR (register) — 64-bit SIMD vector	-
0	10	0001	1	1	VORR (register) — 128-bit SIMD vector	-
0	10	1100		0	SHA1M	FEAT_SHA1
0	11	0001	0	1	VORN (register) — 64-bit SIMD vector	-
0	11	0001	1	1	VORN (register) — 128-bit SIMD vector	-
0	11	1100		0	SHA1SU0	FEAT_SHA1
1		1000	0	0	VSUB (integer) — 64-bit SIMD vector	-
1		1000	0	1	VCEQ (register) — A1 64-bit SIMD vector	-
1		1000	1	0	VSUB (integer) — 128-bit SIMD vector	-
1		1000	1	1	VCEQ (register) — A1 128-bit SIMD vector	-
1		1001	0	0	VMLS (integer) — 64-bit SIMD vector	-
1		1001	1	0	VMLS (integer) — 128-bit SIMD vector	-
1		1011	0	0	VQRDMULH — 64-bit SIMD vector	-
1		1011	0	1	VQRDMLAH — 64-bit SIMD vector	FEAT_RDM
1		1011	1	0	VQRDMULH — 128-bit SIMD vector	-
1		1011	1	1	VQRDMLAH — 128-bit SIMD vector	FEAT_RDM
1		1100	0	1	VQRDMLSH — 64-bit SIMD vector	FEAT_RDM

Decode fields					Instruction Details	Condition
U	size	opc	Q	o1		
1		1100	1	1	VQRDMLSH — 128-bit SIMD vector	FEAT_RDM
1		1111	1	0	UNALLOCATED	-
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101	0	1	VMUL (floating-point) — 64-bit SIMD vector	-
1	0x	1101	1	1	VMUL (floating-point) — 128-bit SIMD vector	-
1	0x	1110	0	0	VCGE (register) — A2 64-bit SIMD vector	-
1	0x	1110	0	1	VACGE — 64-bit SIMD vector	-
1	0x	1110	1	0	VCGE (register) — A2 128-bit SIMD vector	-
1	0x	1110	1	1	VACGE — 128-bit SIMD vector	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111	0	1	VMAXNM — 64-bit SIMD vector	-
1	0x	1111	1	1	VMAXNM — 128-bit SIMD vector	-
1	00	0001	0	1	VEOR — 64-bit SIMD vector	-
1	00	0001	1	1	VEOR — 128-bit SIMD vector	-
1	00	1100		0	SHA256H	FEAT_SHA256
1	01	0001	0	1	VBSL — 64-bit SIMD vector	-
1	01	0001	1	1	VBSL — 128-bit SIMD vector	-
1	01	1100		0	SHA256H2	FEAT_SHA256
1	1x	1101		1	UNALLOCATED	-
1	1x	1101	0	0	VABD (floating-point) — 64-bit SIMD vector	-
1	1x	1101	1	0	VABD (floating-point) — 128-bit SIMD vector	-
1	1x	1110	0	0	VCGT (register) — A2 64-bit SIMD vector	-
1	1x	1110	0	1	VACGT — 64-bit SIMD vector	-
1	1x	1110	1	0	VCGT (register) — A2 128-bit SIMD vector	-
1	1x	1110	1	1	VACGT — 128-bit SIMD vector	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111	0	1	VMINNM — 64-bit SIMD vector	-
1	1x	1111	1	1	VMINNM — 128-bit SIMD vector	-
1	10	0001	0	1	VBIT — 64-bit SIMD vector	-
1	10	0001	1	1	VBIT — 128-bit SIMD vector	-
1	10	1100		0	SHA256SU1	FEAT_SHA256
1	11	0001	0	1	VBIF — 64-bit SIMD vector	-
1	11	0001	1	1	VBIF — 128-bit SIMD vector	-
1	11	1100		0	UNALLOCATED	-

Advanced SIMD two registers, or three registers of different lengths

The encodings in this section are decoded from [Advanced SIMD data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	op0	1			op1										op2					op3		0			

Decode fields				Instruction details
op0	op1	op2	op3	
0	11			Advanced SIMD vector extract
1	11	0x		Advanced SIMD two registers misc
1	11	10		Advanced SIMD table permute
1	11	11		Advanced SIMD duplicate (scalar)
	!= 11		0	Advanced SIMD three registers of different lengths
	!= 11		1	Advanced SIMD two registers and a scalar

Advanced SIMD vector extract

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

Decode fields		Instruction Details
Q		
0		VEXT (byte elements) — 64-bit SIMD vector
1		VEXT (byte elements) — 128-bit SIMD vector

Advanced SIMD two registers misc

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	opc1	Vd			0	opc2			Q	M	0	Vm							

Decode fields				Instruction Details	Condition
size	opc1	opc2	Q		
	00	0000	0	VREV64 — 64-bit SIMD vector	-
	00	0000	1	VREV64 — 128-bit SIMD vector	-
	00	0001	0	VREV32 — 64-bit SIMD vector	-
	00	0001	1	VREV32 — 128-bit SIMD vector	-
	00	0010	0	VREV16 — 64-bit SIMD vector	-
	00	0010	1	VREV16 — 128-bit SIMD vector	-
	00	0011		UNALLOCATED	-
	00	010x	0	VPADDL — 64-bit SIMD vector	-
	00	010x	1	VPADDL — 128-bit SIMD vector	-
	00	0110	0	AESE	FEAT_AES
	00	0110	1	AESD	FEAT_AES
	00	0111	0	AESMC	FEAT_AES
	00	0111	1	AESIMC	FEAT_AES
	00	1000	0	VCLS — 64-bit SIMD vector	-
	00	1000	1	VCLS — 128-bit SIMD vector	-
	00	1001	0	VCLZ — 64-bit SIMD vector	-
	00	1001	1	VCLZ — 128-bit SIMD vector	-
	00	1010	0	VCNT — 64-bit SIMD vector	-
	00	1010	1	VCNT — 128-bit SIMD vector	-
	00	1011	0	VMVN (register) — 64-bit SIMD vector	-
	00	1011	1	VMVN (register) — 128-bit SIMD vector	-
	00	110x	0	VPADAL — 64-bit SIMD vector	-
	00	110x	1	VPADAL — 128-bit SIMD vector	-
	00	1110	0	VQABS — 64-bit SIMD vector	-
	00	1110	1	VQABS — 128-bit SIMD vector	-
	00	1111	0	VQNEG — 64-bit SIMD vector	-
	00	1111	1	VQNEG — 128-bit SIMD vector	-
	01	x000	0	VCGT (immediate #0) — 64-bit SIMD vector	-
	01	x000	1	VCGT (immediate #0) — 128-bit SIMD vector	-
	01	x001	0	VCGE (immediate #0) — 64-bit SIMD vector	-
	01	x001	1	VCGE (immediate #0) — 128-bit SIMD vector	-
	01	x010	0	VCEQ (immediate #0) — 64-bit SIMD vector	-
	01	x010	1	VCEQ (immediate #0) — 128-bit SIMD vector	-
	01	x011	0	VCLE (immediate #0) — 64-bit SIMD vector	-
	01	x011	1	VCLE (immediate #0) — 128-bit SIMD vector	-
	01	x100	0	VCLT (immediate #0) — 64-bit SIMD vector	-
	01	x100	1	VCLT (immediate #0) — 128-bit SIMD vector	-
	01	x110	0	VABS — 64-bit SIMD vector	-
	01	x110	1	VABS — 128-bit SIMD vector	-
	01	x111	0	VNEG — 64-bit SIMD vector	-
	01	x111	1	VNEG — 128-bit SIMD vector	-
	01	0101	0	UNALLOCATED	-
	01	0101	1	SHA1H	FEAT_SHA1
	01	1101		UNALLOCATED	-
	10	x110	1	UNALLOCATED	-

size	Decode fields			Q	Instruction Details	Condition
	opc1	opc2				
	10	0001	0		VTRN — 64-bit SIMD vector	-
	10	0001	1		VTRN — 128-bit SIMD vector	-
	10	0010	0		VUZP — 64-bit SIMD vector	-
	10	0010	1		VUZP — 128-bit SIMD vector	-
	10	0011	0		VZIP — 64-bit SIMD vector	-
	10	0011	1		VZIP — 128-bit SIMD vector	-
	10	0100	0		VMOVN	-
	10	0100	1		VQMOVN , VQMOVUN — Unsigned result	-
	10	0101			VQMOVN , VQMOVUN — Signed result	-
	10	0110	0		VSHLL	-
	10	0111	0		SHA1SU1	FEAT_SHA1
	10	0111	1		SHA256SU0	FEAT_SHA256
	10	1000	0		VRINTN (Advanced SIMD) — 64-bit SIMD vector	-
	10	1000	1		VRINTN (Advanced SIMD) — 128-bit SIMD vector	-
	10	1001	0		VRINTX (Advanced SIMD) — 64-bit SIMD vector	-
	10	1001	1		VRINTX (Advanced SIMD) — 128-bit SIMD vector	-
	10	1010	0		VRINTA (Advanced SIMD) — 64-bit SIMD vector	-
	10	1010	1		VRINTA (Advanced SIMD) — 128-bit SIMD vector	-
	10	1011	0		VRINTZ (Advanced SIMD) — 64-bit SIMD vector	-
	10	1011	1		VRINTZ (Advanced SIMD) — 128-bit SIMD vector	-
	10	1100	0		VCVT (between half-precision and single-precision, Advanced SIMD) — Single-precision to half-precision	-
	10	1101	0		VRINTM (Advanced SIMD) — 64-bit SIMD vector	-
	10	1101	1		VRINTM (Advanced SIMD) — 128-bit SIMD vector	-
	10	1110	0		VCVT (between half-precision and single-precision, Advanced SIMD) — Half-precision to single-precision	-
	10	1111	0		VRINTP (Advanced SIMD) — 64-bit SIMD vector	-
	10	1111	1		VRINTP (Advanced SIMD) — 128-bit SIMD vector	-
	11	000x	0		VCVTA (Advanced SIMD) — 64-bit SIMD vector	-
	11	000x	1		VCVTA (Advanced SIMD) — 128-bit SIMD vector	-
	11	001x	0		VCVTN (Advanced SIMD) — 64-bit SIMD vector	-
	11	001x	1		VCVTN (Advanced SIMD) — 128-bit SIMD vector	-
	11	010x	0		VCVTP (Advanced SIMD) — 64-bit SIMD vector	-
	11	010x	1		VCVTP (Advanced SIMD) — 128-bit SIMD vector	-
	11	011x	0		VCVTM (Advanced SIMD) — 64-bit SIMD vector	-
	11	011x	1		VCVTM (Advanced SIMD) — 128-bit SIMD vector	-
	11	10x0	0		VRECPE — 64-bit SIMD vector	-
	11	10x0	1		VRECPE — 128-bit SIMD vector	-
	11	10x1	0		VRSQRTE — 64-bit SIMD vector	-
	11	10x1	1		VRSQRTE — 128-bit SIMD vector	-
	11	11xx	0		VCVT (between floating-point and integer, Advanced SIMD) — 64-bit SIMD vector	-
	11	11xx	1		VCVT (between floating-point and integer, Advanced SIMD) — 128-bit SIMD vector	-
00	10	0000	0		VSWP — 64-bit SIMD vector	-
00	10	0000	1		VSWP — 128-bit SIMD vector	-
01	10	1100	1		VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16

Decode fields				Instruction Details	Condition
size	opc1	opc2	Q		
!= 00	10	0000		UNALLOCATED	-
!= 01	10	1100	1	UNALLOCATED	-

Advanced SIMD table permute

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1		Vn		Vd					1	0	len	N	op	M	0			Vm		

Decode fields				Instruction Details
size	opc			
0				VTBL, VTBX — VTBL
1				VTBL, VTBX — VTBX

Advanced SIMD duplicate (scalar)

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1		imm4				Vd			1	1		opc		O	M	0		Vm		

Decode fields				Instruction Details
opc		Q		
000		0		VDUP (scalar)
000		1		VDUP (scalar)
!= 000				UNALLOCATED

Advanced SIMD three registers of different lengths

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	!= 11		Vn			Vd			opc			N	0	M	0	Vm						
size																															

size

The following constraints also apply to this encoding: size != '11'

Decode fields		Instruction Details
U	opc	
	0000	VADDL
	0001	VADDW
	0010	VSUBL
	0011	VSUBW
	0101	VABAL
	0111	VABDL (integer)
	1000	VMLAL (integer)
	1010	VMSL (integer)
	11×0	VMULL (integer and polynomial)
0	0100	VADDHN
0	0110	VSUBHN
0	1001	VQDMLAL
0	1011	VQDMSL
0	1101	VQDMULL
0	1111	UNALLOCATED
1	0100	VRADDHN
1	0110	VRSUBHN
1	1××1	UNALLOCATED

Advanced SIMD two registers and a scalar

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	!= 11	Vn				Vd				opc				N	1	M	0	Vm				
size																															

The following constraints also apply to this encoding: size != '11'

Decode fields		Instruction Details	Condition
Q	opc		
	0010	VMLAL (by scalar)	-
	0110	VMLSL (by scalar)	-
	1010	VMULL (by scalar)	-
0	000x	VMLA (by scalar) — 64-bit SIMD vector	-
0	0011	VQDMLAL	-
0	010x	VMLS (by scalar) — 64-bit SIMD vector	-
0	0111	VQDMLSL	-
0	100x	VMUL (by scalar) — 64-bit SIMD vector	-
0	1011	VQDMULL	-
0	1100	VQDMULH — 64-bit SIMD vector	-
0	1101	VQRDMULH — 64-bit SIMD vector	-
0	1110	VQRDMLAH — 64-bit SIMD vector	FEAT_RDM
0	1111	VQRDMLSH — 64-bit SIMD vector	FEAT_RDM
1	0x11	UNALLOCATED	-
1	000x	VMLA (by scalar) — 128-bit SIMD vector	-
1	010x	VMLS (by scalar) — 128-bit SIMD vector	-
1	100x	VMUL (by scalar) — 128-bit SIMD vector	-
1	1011	UNALLOCATED	-
1	1100	VQDMULH — 128-bit SIMD vector	-
1	1101	VQRDMULH — 128-bit SIMD vector	-
1	1110	VQRDMLAH — 128-bit SIMD vector	FEAT_RDM
1	1111	VQRDMLSH — 128-bit SIMD vector	FEAT_RDM

Advanced SIMD shifts and immediate generation

The encodings in this section are decoded from [Advanced SIMD data-processing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1		1		op0																	1				

Decode fields		Instruction details
op0		
000xxxxxxxxxxxx0		Advanced SIMD one register and modified immediate
!= 000xxxxxxxxxxxx0		Advanced SIMD two registers and shift amount

Advanced SIMD one register and modified immediate

The encodings in this section are decoded from [Advanced SIMD shifts and immediate generation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd			cmode				0	Q	op	1	imm4				

Decode fields			Instruction Details
cmode	Q	op	
0xx0	0	0	VMOV (immediate) — A1 64-bit SIMD vector
0xx0	0	1	VMVN (immediate) — A1 64-bit SIMD vector
0xx0	1	0	VMOV (immediate) — A1 128-bit SIMD vector
0xx0	1	1	VMVN (immediate) — A1 128-bit SIMD vector
0xx1	0	0	VORR (immediate) — A1 64-bit SIMD vector
0xx1	0	1	VBIC (immediate) — A1 64-bit SIMD vector
0xx1	1	0	VORR (immediate) — A1 128-bit SIMD vector
0xx1	1	1	VBIC (immediate) — A1 128-bit SIMD vector
10x0	0	0	VMOV (immediate) — A3 64-bit SIMD vector
10x0	0	1	VMVN (immediate) — A2 64-bit SIMD vector
10x0	1	0	VMOV (immediate) — A3 128-bit SIMD vector
10x0	1	1	VMVN (immediate) — A2 128-bit SIMD vector
10x1	0	0	VORR (immediate) — A2 64-bit SIMD vector
10x1	0	1	VBIC (immediate) — A2 64-bit SIMD vector
10x1	1	0	VORR (immediate) — A2 128-bit SIMD vector
10x1	1	1	VBIC (immediate) — A2 128-bit SIMD vector
11xx	0	0	VMOV (immediate) — A4 64-bit SIMD vector
11xx	1	0	VMOV (immediate) — A4 128-bit SIMD vector
110x	0	1	VMVN (immediate) — A3 64-bit SIMD vector
110x	1	1	VMVN (immediate) — A3 128-bit SIMD vector
1110	0	1	VMOV (immediate) — A5 64-bit SIMD vector
1110	1	1	VMOV (immediate) — A5 128-bit SIMD vector
1111		1	UNALLOCATED

Advanced SIMD two registers and shift amount

The encodings in this section are decoded from [Advanced SIMD shifts and immediate generation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm3H			imm3L			Vd			opc			L		Q	M	1	Vm				

The following constraints also apply to this encoding: `!(imm3H == '000' && L == '0')`

Decode fields						Instruction Details
U	imm3H	imm3L	opc	L	Q	
			1xxx	1		UNALLOCATED
	000		0000	1	0	VSHR — 64-bit SIMD vector
	000		0000	1	1	VSHR — 128-bit SIMD vector
	000		0001	1	0	VSRA — 64-bit SIMD vector
	000		0001	1	1	VSRA — 128-bit SIMD vector
	000		0010	1	0	VRSHR — 64-bit SIMD vector
	000		0010	1	1	VRSHR — 128-bit SIMD vector
	000		0011	1	0	VRSRA — 64-bit SIMD vector
	000		0011	1	1	VRSRA — 128-bit SIMD vector
	000		0111	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
	000		0111	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
	001	000	1010	0	0	VMOVL
	001	!= 000	1010	0	0	VSHLL
	010	000	1010	0	0	VMOVL
	010	!= 000	1010	0	0	VSHLL
	011		1010	0	0	VSHLL
	100	000	1010	0	0	VMOVL
	100	!= 000	1010	0	0	VSHLL
	101		1010	0	0	VSHLL
	11x		1010	0	0	VSHLL
	!= 000		0000		0	VSHR — 64-bit SIMD vector
	!= 000		0000		1	VSHR — 128-bit SIMD vector
	!= 000		0001		0	VSRA — 64-bit SIMD vector
	!= 000		0001		1	VSRA — 128-bit SIMD vector
	!= 000		0010		0	VRSHR — 64-bit SIMD vector
	!= 000		0010		1	VRSHR — 128-bit SIMD vector
	!= 000		0011		0	VRSRA — 64-bit SIMD vector
	!= 000		0011		1	VRSRA — 128-bit SIMD vector
	!= 000		0111		0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
	!= 000		0111		1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
	!= 000		1001	0	0	VQSHRN, VQSHRUN — Signed result
	!= 000		1001	0	1	VQRSHRN, VQRSHRUN — Signed result

Decode fields						Instruction Details
U	imm3H	imm3L	opc	L	Q	
	!= 000		1010	0	1	UNALLOCATED
	!= 000		1011	0		UNALLOCATED
	!= 000		11xx	0	0	VCVT (between floating-point and fixed-point, Advanced SIMD) — 64-bit SIMD vector
	!= 000		11xx	0	1	VCVT (between floating-point and fixed-point, Advanced SIMD) — 128-bit SIMD vector
0			01x0	1		UNALLOCATED
0	000		0101	1	0	VSHL (immediate) — 64-bit SIMD vector
0	000		0101	1	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000		01x0	0		UNALLOCATED
0	!= 000		0101		0	VSHL (immediate) — 64-bit SIMD vector
0	!= 000		0101		1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000		1000	0	0	VSHRN
0	!= 000		1000	0	1	VRSHRN
1	000		0100	1	0	VSRI — 64-bit SIMD vector
1	000		0100	1	1	VSRI — 128-bit SIMD vector
1	000		0101	1	0	VSLI — 64-bit SIMD vector
1	000		0101	1	1	VSLI — 128-bit SIMD vector
1	000		0110	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	000		0110	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000		0100		0	VSRI — 64-bit SIMD vector
1	!= 000		0100		1	VSRI — 128-bit SIMD vector
1	!= 000		0101		0	VSLI — 64-bit SIMD vector
1	!= 000		0101		1	VSLI — 128-bit SIMD vector
1	!= 000		0110		0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	!= 000		0110		1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000		1000	0	0	VQSHRN, VQSHRUN — Unsigned result
1	!= 000		1000	0	1	VQRSHRN, VQRSHRUN — Unsigned result

Advanced SIMD element or structure load/store

The encodings in this section are decoded from [Unconditional instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	op0			0										op1										

Decode fields		Instruction details
op0	op1	
0		Advanced SIMD load/store multiple structures
1	1 1	Advanced SIMD load single structure to all lanes
1	!= 1 1	Advanced SIMD load/store single structure to one lane

Advanced SIMD load/store multiple structures

The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	L	0	Rn				Vd				itype				size		align		Rm			

Decode fields			Instruction Details
L	itype	Rm	
	1011		UNALLOCATED
	11xx		UNALLOCATED
0	000x	1101	VST4 (multiple 4-element structures) — Post-indexed
0	000x	1111	VST4 (multiple 4-element structures) — Offset
0	000x	!= 11x1	VST4 (multiple 4-element structures) — Post-indexed
0	0010	1101	VST1 (multiple single elements) — A4 Post-indexed
0	0010	1111	VST1 (multiple single elements) — A4 Offset
0	0010	!= 11x1	VST1 (multiple single elements) — A4 Post-indexed
0	0011	1101	VST2 (multiple 2-element structures) — A2 Post-indexed
0	0011	1111	VST2 (multiple 2-element structures) — A2 Offset
0	0011	!= 11x1	VST2 (multiple 2-element structures) — A2 Post-indexed
0	010x	1101	VST3 (multiple 3-element structures) — Post-indexed
0	010x	1111	VST3 (multiple 3-element structures) — Offset
0	010x	!= 11x1	VST3 (multiple 3-element structures) — Post-indexed
0	0110	1101	VST1 (multiple single elements) — A3 Post-indexed
0	0110	1111	VST1 (multiple single elements) — A3 Offset
0	0110	!= 11x1	VST1 (multiple single elements) — A3 Post-indexed
0	0111	1101	VST1 (multiple single elements) — A1 Post-indexed
0	0111	1111	VST1 (multiple single elements) — A1 Offset
0	0111	!= 11x1	VST1 (multiple single elements) — A1 Post-indexed
0	100x	1101	VST2 (multiple 2-element structures) — A1 Post-indexed
0	100x	1111	VST2 (multiple 2-element structures) — A1 Offset
0	100x	!= 11x1	VST2 (multiple 2-element structures) — A1 Post-indexed
0	1010	1101	VST1 (multiple single elements) — A2 Post-indexed
0	1010	1111	VST1 (multiple single elements) — A2 Offset
0	1010	!= 11x1	VST1 (multiple single elements) — A2 Post-indexed
1	000x	1101	VLD4 (multiple 4-element structures) — Post-indexed
1	000x	1111	VLD4 (multiple 4-element structures) — Offset
1	000x	!= 11x1	VLD4 (multiple 4-element structures) — Post-indexed
1	0010	1101	VLD1 (multiple single elements) — A4 Post-indexed
1	0010	1111	VLD1 (multiple single elements) — A4 Offset
1	0010	!= 11x1	VLD1 (multiple single elements) — A4 Post-indexed
1	0011	1101	VLD2 (multiple 2-element structures) — A2 Post-indexed
1	0011	1111	VLD2 (multiple 2-element structures) — A2 Offset
1	0011	!= 11x1	VLD2 (multiple 2-element structures) — A2 Post-indexed
1	010x	1101	VLD3 (multiple 3-element structures) — Post-indexed
1	010x	1111	VLD3 (multiple 3-element structures) — Offset
1	010x	!= 11x1	VLD3 (multiple 3-element structures) — Post-indexed
1	0110	1101	VLD1 (multiple single elements) — A3 Post-indexed
1	0110	1111	VLD1 (multiple single elements) — A3 Offset
1	0110	!= 11x1	VLD1 (multiple single elements) — A3 Post-indexed
1	0111	1101	VLD1 (multiple single elements) — A1 Post-indexed
1	0111	1111	VLD1 (multiple single elements) — A1 Offset
1	0111	!= 11x1	VLD1 (multiple single elements) — A1 Post-indexed
1	100x	1101	VLD2 (multiple 2-element structures) — A1 Post-indexed

Decode fields			Instruction Details
L	itype	Rm	
1	100x	1111	VLD2 (multiple 2-element structures) — A1 Offset
1	100x	!= 11x1	VLD2 (multiple 2-element structures) — A1 Post-indexed
1	1010	1101	VLD1 (multiple single elements) — A2 Post-indexed
1	1010	1111	VLD1 (multiple single elements) — A2 Offset
1	1010	!= 11x1	VLD1 (multiple single elements) — A2 Post-indexed

Advanced SIMD load single structure to all lanes

The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	L	0	Rn				Vd				1	1	N	size		T	a	Rm				

Decode fields				Instruction Details
L	N	a	Rm	
0				UNALLOCATED
1	00		1101	VLD1 (single element to all lanes) — Post-indexed
1	00		1111	VLD1 (single element to all lanes) — Offset
1	00		!= 11x1	VLD1 (single element to all lanes) — Post-indexed
1	01		1101	VLD2 (single 2-element structure to all lanes) — Post-indexed
1	01		1111	VLD2 (single 2-element structure to all lanes) — Offset
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes) — Post-indexed
1	10	0	1101	VLD3 (single 3-element structure to all lanes) — Post-indexed
1	10	0	1111	VLD3 (single 3-element structure to all lanes) — Offset
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes) — Post-indexed
1	10	1		UNALLOCATED
1	11		1101	VLD4 (single 4-element structure to all lanes) — Post-indexed
1	11		1111	VLD4 (single 4-element structure to all lanes) — Offset
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes) — Post-indexed

Advanced SIMD load/store single structure to one lane

The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	L	0	Rn				Vd				!= 11		N	index		align	Rm					
size																															

size

The following constraints also apply to this encoding: size != '11'

Decode fields				Instruction Details
L	size	N	Rm	
0	00	00	1101	VST1 (single element from one lane) — A1 Post-indexed
0	00	00	1111	VST1 (single element from one lane) — A1 Offset
0	00	00	!= 11x1	VST1 (single element from one lane) — A1 Post-indexed
0	00	01	1101	VST2 (single 2-element structure from one lane) — A1 Post-indexed
0	00	01	1111	VST2 (single 2-element structure from one lane) — A1 Offset
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane) — A1 Post-indexed
0	00	10	1101	VST3 (single 3-element structure from one lane) — A1 Post-indexed
0	00	10	1111	VST3 (single 3-element structure from one lane) — A1 Offset
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane) — A1 Post-indexed
0	00	11	1101	VST4 (single 4-element structure from one lane) — A1 Post-indexed
0	00	11	1111	VST4 (single 4-element structure from one lane) — A1 Offset
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane) — A1 Post-indexed
0	01	00	1101	VST1 (single element from one lane) — A2 Post-indexed
0	01	00	1111	VST1 (single element from one lane) — A2 Offset
0	01	00	!= 11x1	VST1 (single element from one lane) — A2 Post-indexed
0	01	01	1101	VST2 (single 2-element structure from one lane) — A2 Post-indexed
0	01	01	1111	VST2 (single 2-element structure from one lane) — A2 Offset
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane) — A2 Post-indexed
0	01	10	1101	VST3 (single 3-element structure from one lane) — A2 Post-indexed
0	01	10	1111	VST3 (single 3-element structure from one lane) — A2 Offset
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane) — A2 Post-indexed
0	01	11	1101	VST4 (single 4-element structure from one lane) — A2 Post-indexed
0	01	11	1111	VST4 (single 4-element structure from one lane) — A2 Offset
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane) — A2 Post-indexed
0	10	00	1101	VST1 (single element from one lane) — A3 Post-indexed
0	10	00	1111	VST1 (single element from one lane) — A3 Offset
0	10	00	!= 11x1	VST1 (single element from one lane) — A3 Post-indexed
0	10	01	1101	VST2 (single 2-element structure from one lane) — A3 Post-indexed
0	10	01	1111	VST2 (single 2-element structure from one lane) — A3 Offset
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane) — A3 Post-indexed
0	10	10	1101	VST3 (single 3-element structure from one lane) — A3 Post-indexed
0	10	10	1111	VST3 (single 3-element structure from one lane) — A3 Offset
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane) — A3 Post-indexed
0	10	11	1101	VST4 (single 4-element structure from one lane) — A3 Post-indexed
0	10	11	1111	VST4 (single 4-element structure from one lane) — A3 Offset
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane) — A3 Post-indexed
1	00	00	1101	VLD1 (single element to one lane) — A1 Post-indexed
1	00	00	1111	VLD1 (single element to one lane) — A1 Offset
1	00	00	!= 11x1	VLD1 (single element to one lane) — A1 Post-indexed
1	00	01	1101	VLD2 (single 2-element structure to one lane) — A1 Post-indexed
1	00	01	1111	VLD2 (single 2-element structure to one lane) — A1 Offset
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane) — A1 Post-indexed
1	00	10	1101	VLD3 (single 3-element structure to one lane) — A1 Post-indexed
1	00	10	1111	VLD3 (single 3-element structure to one lane) — A1 Offset
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane) — A1 Post-indexed

Decode fields				Instruction Details
L	size	N	Rm	
1	00	11	1101	VLD4 (single 4-element structure to one lane) — A1 Post-indexed
1	00	11	1111	VLD4 (single 4-element structure to one lane) — A1 Offset
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane) — A1 Post-indexed
1	01	00	1101	VLD1 (single element to one lane) — A2 Post-indexed
1	01	00	1111	VLD1 (single element to one lane) — A2 Offset
1	01	00	!= 11x1	VLD1 (single element to one lane) — A2 Post-indexed
1	01	01	1101	VLD2 (single 2-element structure to one lane) — A2 Post-indexed
1	01	01	1111	VLD2 (single 2-element structure to one lane) — A2 Offset
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane) — A2 Post-indexed
1	01	10	1101	VLD3 (single 3-element structure to one lane) — A2 Post-indexed
1	01	10	1111	VLD3 (single 3-element structure to one lane) — A2 Offset
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane) — A2 Post-indexed
1	01	11	1101	VLD4 (single 4-element structure to one lane) — A2 Post-indexed
1	01	11	1111	VLD4 (single 4-element structure to one lane) — A2 Offset
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane) — A2 Post-indexed
1	10	00	1101	VLD1 (single element to one lane) — A3 Post-indexed
1	10	00	1111	VLD1 (single element to one lane) — A3 Offset
1	10	00	!= 11x1	VLD1 (single element to one lane) — A3 Post-indexed
1	10	01	1101	VLD2 (single 2-element structure to one lane) — A3 Post-indexed
1	10	01	1111	VLD2 (single 2-element structure to one lane) — A3 Offset
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane) — A3 Post-indexed
1	10	10	1101	VLD3 (single 3-element structure to one lane) — A3 Post-indexed
1	10	10	1111	VLD3 (single 3-element structure to one lane) — A3 Offset
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane) — A3 Post-indexed
1	10	11	1101	VLD4 (single 4-element structure to one lane) — A3 Post-indexed
1	10	11	1111	VLD4 (single 4-element structure to one lane) — A3 Offset
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane) — A3 Post-indexed

Memory hints and barriers

The encodings in this section are decoded from [Unconditional instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	op0					1																	op1			

Decode fields		Instruction details	
op0	op1		
00xx1		CONSTRAINED UNPREDICTABLE	
01001		CONSTRAINED UNPREDICTABLE	
01011		Barriers	
011x1		CONSTRAINED UNPREDICTABLE	
0xxx0		Preload (immediate)	
1xxx0	0	Preload (register)	
1xxx1	0	CONSTRAINED UNPREDICTABLE	
1xxxx	1	UNALLOCATED	

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

Barriers

The encodings in this section are decoded from [Memory hints and barriers](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	opcode				option			

Decode fields		Instruction Details		Condition
opcode	option			
0000		CONSTRAINED UNPREDICTABLE		-
0001		CLREX		-
001x		CONSTRAINED UNPREDICTABLE		-
0100	0000	SSBB		-
0100	0100	PSSBB		-
0100	!= 0x00	DSB		-
0101		DMB		-
0110		ISB		-
0111		SB		FEAT_SB
1xxx		CONSTRAINED UNPREDICTABLE		-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

Preload (immediate)

The encodings in this section are decoded from [Memory hints and barriers](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	D	U	R	0	1	Rn				(1)	(1)	(1)	(1)	imm12											

Decode fields			Instruction Details	
D	R	Rn		
0	0		Reserved hint, behaves as NOP	
0	1		PLI (immediate, literal)	
1		1111	PLD (literal)	
1	0	!= 1111	PLD, PLDW (immediate) — Preload write	
1	1	!= 1111	PLD, PLDW (immediate) — Preload read	

Preload (register)

The encodings in this section are decoded from [Memory hints and barriers](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	D	U	o2	0	1	Rn				(1)	(1)	(1)	(1)	imm5				stype		0	Rm				

Decode fields			Instruction Details	
D	o2	imm5:stype		
0	0		Reserved hint, behaves as NOP	
0	1	0000011	PLI (register) — Rotate right with extend	
0	1	!= 0000011	PLI (register) — Shift or rotate by value	
1	0	0000011	PLD, PLDW (register) — Preload write, rotate right with extend	
1	0	!= 0000011	PLD, PLDW (register) — Preload write, optional shift or rotate	
1	1	0000011	PLD, PLDW (register) — Preload read, rotate right with extend	
1	1	!= 0000011	PLD, PLDW (register) — Preload read, optional shift or rotate	

T32 instruction set encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0					op1																										

Decode fields

Instruction details

op0	op1	
111	00	B — T2
!= 111		16-bit
111	!= 00	32-bit

16-bit

The encodings in this section are decoded from [T32 instruction set encoding](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 111xxx															
op0															

The following constraints also apply to this encoding: !(op0 IN {'111xxx'})

Decode fields

Instruction details

op0	
00xxxxx	Shift (immediate), add, subtract, move, and compare
010000	Data-processing (two low registers)
010001	Special data instructions and branch and exchange
01001x	LDR (literal) — T1
0101xx	Load/store (register offset)
011xxx	Load/store word/byte (immediate offset)
1000xx	Load/store halfword (immediate offset)
1001xx	Load/store (SP-relative)
1010xx	Add PC/SP (immediate)
1011xx	Miscellaneous 16-bit instructions
1100xx	Load/store multiple
1101xx	Conditional branch, and Supervisor Call

Shift (immediate), add, subtract, move, and compare

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	op0	op1	op2											

Decode fields

Instruction details

op0	op1	op2	
0	11	0	Add, subtract (three low registers)
0	11	1	Add, subtract (two low registers and immediate)
0	!= 11		MOV, MOVS (register) — T2
1			Add, subtract, compare, move (one low register and immediate)

Add, subtract (three low registers)

The encodings in this section are decoded from [Shift \(immediate\), add, subtract, move, and compare](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	S	Rm			Rn			Rd		

Decode fields

S

Instruction Details

0	ADD, ADDS (register)
1	SUB, SUBS (register)

Add, subtract (two low registers and immediate)

The encodings in this section are decoded from [Shift \(immediate\), add, subtract, move, and compare](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	S	imm3			Rn			Rd		

Decode fields

S

Instruction Details

0	ADD, ADDS (immediate)
1	SUB, SUBS (immediate)

Add, subtract, compare, move (one low register and immediate)

The encodings in this section are decoded from [Shift \(immediate\), add, subtract, move, and compare](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	op		Rd		imm8								

Decode fields

op

Instruction Details

00	MOV, MOVS (immediate)
01	CMP (immediate)
10	ADD, ADDS (immediate)
11	SUB, SUBS (immediate)

Data-processing (two low registers)

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	op				Rs			Rd		

Decode fields

op

Instruction Details

0000	AND, ANDS (register)
0001	EOR, EORS (register)
0010	MOV, MOVS (register-shifted register) — Logical shift left
0011	MOV, MOVS (register-shifted register) — Logical shift right
0100	MOV, MOVS (register-shifted register) — Arithmetic shift right
0101	ADC, ADCS (register)
0110	SBC, SBCS (register)
0111	MOV, MOVS (register-shifted register) — Rotate right
1000	TST (register)
1001	RSB, RSBS (immediate)
1010	CMP (register)
1011	CMN (register)
1100	ORR, ORRS (register)
1101	MUL, MULS
1110	BIC, BICS (register)
1111	MVN, MVNS (register)

Special data instructions and branch and exchange

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	op0									

Decode fields

op0

Instruction details

11	Branch and exchange
!= 11	Add, subtract, compare, move (two high registers)

Branch and exchange

The encodings in this section are decoded from [Special data instructions and branch and exchange](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	L		Rm			(0)	(0)	(0)

Decode fields

L

Instruction Details

0	BX
1	BLX (register)

Add, subtract, compare, move (two high registers)

The encodings in this section are decoded from [Special data instructions and branch and exchange](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	!= 11	D		Rs				Rd		
						op									

The following constraints also apply to this encoding: op != '11'

Decode fields			Instruction Details
op	D:Rd	Rs	
00		1101	ADD, ADDS (SP plus register) — T1
00	1101	!= 1101	ADD, ADDS (SP plus register) — T2
00	!= 1101	!= 1101	ADD, ADDS (register)
01			CMP (register)
10			MOV, MOVS (register)

Load/store (register offset)

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	L	B	H	Rm			Rn			Rt		

Decode fields			Instruction Details
L	B	H	
0	0	0	STR (register)
0	0	1	STRH (register)
0	1	0	STRB (register)
0	1	1	LDRSB (register)
1	0	0	LDR (register)
1	0	1	LDRH (register)
1	1	0	LDRB (register)
1	1	1	LDRSH (register)

Load/store word/byte (immediate offset)

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	B	L	imm5					Rn			Rt		

Decode fields		Instruction Details
B	L	
0	0	STR (immediate)
0	1	LDR (immediate)
1	0	STRB (immediate)
1	1	LDRB (immediate)

Load/store halfword (immediate offset)

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	L	imm5					Rn			Rt		

Decode fields		Instruction Details
L		
0		STRH (immediate)
1		LDRH (immediate)

Load/store (SP-relative)

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	L	Rt			imm8							

Decode fields

L

Instruction Details

0	STR (immediate)
1	LDR (immediate)

Add PC/SP (immediate)

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	SP		Rd									imm8

Decode fields

SP

Instruction Details

0	ADR
1	ADD, ADDS (SP plus immediate)

Miscellaneous 16-bit instructions

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1		op0			op1	op2						op3

Decode fields

op0

op1

op2

op3

Instruction details

Condition

0000				Adjust SP (immediate)	-
0010				Extend	-
0110	00	0		SETPAN	FEAT_PAN
0110	00	1		UNALLOCATED	-
0110	01			Change Processor State	-
0110	1x			UNALLOCATED	-
0111				UNALLOCATED	-
1000				UNALLOCATED	-
1010	10			HLT	-
1010	!= 10			Reverse bytes	-
1110				BKPT	-
1111			0000	Hints	-
1111			!= 0000	IT	-
x0x1				Compare and branch zero/non-zero	-
x10x				Push and Pop	-

Adjust SP (immediate)

The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	S							imm7

Decode fields

S

Instruction Details

0	ADD, ADDS (SP plus immediate)
1	SUB, SUBS (SP minus immediate)

Extend

The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	U	B	Rm			Rd		

Decode fields		Instruction Details
U	B	
0	0	SXTH
0	1	SXTB
1	0	UXTH
1	1	UXTB

Change Processor State

The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	op	flags				

Decode fields		Instruction Details
op	flags	
0		SETEND
1	0xxxx	CPS, CPSID, CPSIE — Interrupt enable
1	1xxxx	CPS, CPSID, CPSIE — Interrupt disable

Reverse bytes

The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	!= 10			Rm		Rd		
op															

The following constraints also apply to this encoding: op != '10'

Decode fields		Instruction Details
op		
00		REV
01		REV16
11		REVSH

Hints

The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	hint				0	0	0	0

Decode fields		Instruction Details
hint		
0000		NOP
0001		YIELD
0010		WFE
0011		WFI
0100		SEV
0101		SEVL
011x		Reserved hint, behaves as NOP
1xxx		Reserved hint, behaves as NOP

Compare and branch zero/non-zero

The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	op	0	i	1	imm5					Rn		

Decode fields	Instruction Details
op	
0	CBNZ, CBZ — CBZ
1	CBNZ, CBZ — CBNZ

Push and Pop

The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	L	1	0	P	register list							

Decode fields	Instruction Details
L	
0	PUSH
1	POP

Load/store multiple

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	L	Rn			register list							

Decode fields	Instruction Details
L	
0	STM, STMIA, STMEA
1	LDM, LDMIA, LDMFD

Conditional branch, and Supervisor Call

The encodings in this section are decoded from [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	op0											

Decode fields	Instruction details
op0	
111x	Exception generation
!= 111x	B — T1

Exception generation

The encodings in this section are decoded from [Conditional branch, and Supervisor Call](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	S	imm8							

Decode fields	Instruction Details
S	
0	UDF
1	SVC

32-bit

The encodings in this section are decoded from [T32 instruction set encoding](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	!= 00xx				op1								op3																
op0																															

The following constraints also apply to this encoding: !(op0 IN {'00xx'})

Decode fields			Instruction details
op0	op1	op3	
0100	xx0xx		Load/store multiple
0100	xx1xx		Load/store dual, load/store exclusive, load-acquire/store-release, and table branch
0101			Data-processing (shifted register)
10x0		0	Data-processing (modified immediate)
10x1	xxxx0	0	Data-processing (plain binary immediate)
10x1	xxxx1	0	UNALLOCATED
10xx		1	Branches and miscellaneous control
1100	1xxx0		Advanced SIMD element or structure load/store
1100	!= 1xxx0		Load/store single
1101	0xxxx		Data-processing (register)
1101	10xxx		Multiply, multiply accumulate, and absolute difference
1101	11xxx		Long multiply and divide
x11x			System register access, Advanced SIMD, and floating-point

Load/store multiple

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	opc		0	W	L	Rn				P	M	register_list													

Decode fields		Instruction Details
opc	L	
00	0	SRS, SRSDA, SRSDb, SRSIA, SRSIB — T1
00	1	RFE, RFEDA, RFEDb, RFEIA, RFEIB — T1
01	0	STM, STMIA, STMEA
01	1	LDM, LDMIA, LDMFD
10	0	STMDB, STMFD
10	1	LDMDB, LDMEA
11	0	SRS, SRSDA, SRSDb, SRSIA, SRSIB — T2
11	1	RFE, RFEDA, RFEDb, RFEIA, RFEIB — T2

Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	op0				op1	op2								op3											

Load-acquire / Store-release

The encodings in this section are decoded from [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	L	Rn				Rt				Rt2				1	op	sz		Rd			
op0																															

Decode fields			Instruction Details
L	op	sz	
	0	11	UNALLOCATED
0	0	00	STLB
0	0	01	STLH
0	0	10	STL
0	1	00	STLEXB
0	1	01	STLEXH
0	1	10	STLEX
0	1	11	STLEXD
1	0	00	LDAB
1	0	01	LDAH
1	0	10	LDA
1	1	00	LDAEXB
1	1	01	LDAEXH
1	1	10	LDAEX
1	1	11	LDAEXD

Load dual (literal)

The encodings in this section are decoded from [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	P	U	1	W	L	1	1	1	1	Rt			Rt2			imm8									

The following constraints also apply to this encoding: !(P == '0' && W == '0')

Decode fields		Instruction Details
L		
0		UNALLOCATED
1		LDRD (literal)

Load/store dual (immediate, post-indexed)

The encodings in this section are decoded from [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	U	1	1	L	!= 1111				Rt			Rt2			imm8									
P								W				Rn																			

The following constraints also apply to this encoding: Rn != '1111'

Decode fields		Instruction Details
L		
0		STRD (immediate)
1		LDRD (immediate)

Load/store dual (immediate)

The encodings in this section are decoded from [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	U	1	0	L	!= 1111				Rt				Rt2				imm8							
							P				W	Rn																			

The following constraints also apply to this encoding: Rn != '1111'

Decode fields																Instruction Details															
L																															
0																STRD (immediate)															
1																LDRD (immediate)															

Load/store dual (immediate, pre-indexed)

The encodings in this section are decoded from [Load/store dual, load/store exclusive, load-acquire/store-release, and table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	U	1	1	L	!= 1111				Rt				Rt2				imm8							
							P				W	Rn																			

The following constraints also apply to this encoding: Rn != '1111'

Decode fields																Instruction Details															
L																															
0																STRD (immediate)															
1																LDRD (immediate)															

Data-processing (shifted register)

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	op1				S	Rn			(0)	imm3			Rd			imm2		stype		Rm					

Decode fields					Instruction Details
op1	S	Rn	imm3:imm2:type	Rd	
0000	0		0000011		AND, ANDS (register) — AND, rotate right with extend
0000	0		!= 0000011		AND, ANDS (register) — AND, shift or rotate by value
0000	1		0000011	1111	TST (register) — Rotate right with extend
0000	1		0000011	!= 1111	AND, ANDS (register) — ANDS, rotate right with extend
0000	1		!= 0000011	1111	TST (register) — Shift or rotate by value
0000	1		!= 0000011	!= 1111	AND, ANDS (register) — ANDS, shift or rotate by value
0001	0		0000011		BIC, BICS (register) — BIC, rotate right with extend
0001	0		!= 0000011		BIC, BICS (register) — BIC, shift or rotate by value
0001	1		0000011		BIC, BICS (register) — BICS, rotate right with extend
0001	1		!= 0000011		BIC, BICS (register) — BICS, shift or rotate by value
0010	0	1111	0000011		MOV, MOVS (register) — MOV, rotate right with extend
0010	0	1111	!= 0000011		MOV, MOVS (register) — MOV, shift or rotate by value
0010	0	!= 1111	0000011		ORR, ORRS (register) — ORR, rotate right with extend
0010	0	!= 1111	!= 0000011		ORR, ORRS (register) — ORR, shift or rotate by value
0010	1	1111	0000011		MOV, MOVS (register) — MOVS, rotate right with extend
0010	1	1111	!= 0000011		MOV, MOVS (register) — MOVS, shift or rotate by value
0010	1	!= 1111	0000011		ORR, ORRS (register) — ORRS, rotate right with extend
0010	1	!= 1111	!= 0000011		ORR, ORRS (register) — ORRS, shift or rotate by value
0011	0	1111	0000011		MVN, MVNS (register) — MVN, rotate right with extend
0011	0	1111	!= 0000011		MVN, MVNS (register) — MVN, shift or rotate by value
0011	0	!= 1111	0000011		ORN, ORNS (register) — ORN, rotate right with extend
0011	0	!= 1111	!= 0000011		ORN, ORNS (register) — ORN, shift or rotate by value
0011	1	1111	0000011		MVN, MVNS (register) — MVNS, rotate right with extend
0011	1	1111	!= 0000011		MVN, MVNS (register) — MVNS, shift or rotate by value
0011	1	!= 1111	0000011		ORN, ORNS (register) — ORNS, rotate right with extend
0011	1	!= 1111	!= 0000011		ORN, ORNS (register) — ORNS, shift or rotate by value
01x1					UNALLOCATED
0100	0		0000011		EOR, EORS (register) — EOR, rotate right with extend
0100	0		!= 0000011		EOR, EORS (register) — EOR, shift or rotate by value
0100	1		0000011	1111	TEQ (register) — Rotate right with extend
0100	1		0000011	!= 1111	EOR, EORS (register) — EORS, rotate right with extend
0100	1		!= 0000011	1111	TEQ (register) — Shift or rotate by value
0100	1		!= 0000011	!= 1111	EOR, EORS (register) — EORS, shift or rotate by value
0110	0		xxxxxx1		UNALLOCATED
0110	0		xxxxxx00		PKHBT, PKHTB — PKHBT
0110	0		xxxxxx10		PKHBT, PKHTB — PKHTB
0110	1				UNALLOCATED
1000	0	1101	0000011		ADD, ADDS (SP plus register) — ADD, rotate right with extend
1000	0	1101	!= 0000011		ADD, ADDS (SP plus register) — ADD, shift or rotate by value
1000	0	!= 1101	0000011		ADD, ADDS (register) — ADD, rotate right with extend
1000	0	!= 1101	!= 0000011		ADD, ADDS (register) — ADD, shift or rotate by value
1000	1		0000011	1111	CMN (register) — Rotate right with extend
1000	1		!= 0000011	1111	CMN (register) — Shift or rotate by value
1000	1	1101	0000011	!= 1111	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
1000	1	1101	!= 0000011	!= 1111	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value

Decode fields					Instruction Details
op1	S	Rn	imm3:imm2:type	Rd	
1000	1	!= 1101	0000011	!= 1111	ADD, ADDS (register) — ADDS, rotate right with extend
1000	1	!= 1101	!= 0000011	!= 1111	ADD, ADDS (register) — ADDS, shift or rotate by value
1001					UNALLOCATED
1010	0		0000011		ADC, ADCS (register) — ADC, rotate right with extend
1010	0		!= 0000011		ADC, ADCS (register) — ADC, shift or rotate by value
1010	1		0000011		ADC, ADCS (register) — ADCS, rotate right with extend
1010	1		!= 0000011		ADC, ADCS (register) — ADCS, shift or rotate by value
1011	0		0000011		SBC, SBCS (register) — SBC, rotate right with extend
1011	0		!= 0000011		SBC, SBCS (register) — SBC, shift or rotate by value
1011	1		0000011		SBC, SBCS (register) — SBCS, rotate right with extend
1011	1		!= 0000011		SBC, SBCS (register) — SBCS, shift or rotate by value
1100					UNALLOCATED
1101	0	1101	0000011		SUB, SUBS (SP minus register) — SUB, rotate right with extend
1101	0	1101	!= 0000011		SUB, SUBS (SP minus register) — SUB, shift or rotate by value
1101	0	!= 1101	0000011		SUB, SUBS (register) — SUB, rotate right with extend
1101	0	!= 1101	!= 0000011		SUB, SUBS (register) — SUB, shift or rotate by value
1101	1		0000011	1111	CMP (register) — Rotate right with extend
1101	1		!= 0000011	1111	CMP (register) — Shift or rotate by value
1101	1	1101	0000011	!= 1111	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
1101	1	1101	!= 0000011	!= 1111	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
1101	1	!= 1101	0000011	!= 1111	SUB, SUBS (register) — SUBS, rotate right with extend
1101	1	!= 1101	!= 0000011	!= 1111	SUB, SUBS (register) — SUBS, shift or rotate by value
1110	0		0000011		RSB, RSBS (register) — RSB, rotate right with extend
1110	0		!= 0000011		RSB, RSBS (register) — RSB, shift or rotate by value
1110	1		0000011		RSB, RSBS (register) — RSBS, rotate right with extend
1110	1		!= 0000011		RSB, RSBS (register) — RSBS, shift or rotate by value
1111					UNALLOCATED

Data-processing (modified immediate)

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	op1		S		Rn		0		imm3		Rd		imm8												

		Decode fields		Instruction Details	
op1	S	Rn	Rd		
0000	0			AND, ANDS (immediate) — AND	
0000	1		1111	TST (immediate)	
0000	1		!= 1111	AND, ANDS (immediate) — ANDS	
0001	0			BIC, BICS (immediate) — BIC	
0001	1			BIC, BICS (immediate) — BICS	
0010	0	1111		MOV, MOVS (immediate) — MOV	
0010	0	!= 1111		ORR, ORRS (immediate) — ORR	
0010	1	1111		MOV, MOVS (immediate) — MOVS	
0010	1	!= 1111		ORR, ORRS (immediate) — ORRS	
0011	0	1111		MVN, MVNS (immediate) — MVN	
0011	0	!= 1111		ORN, ORNS (immediate) — Not flag setting	
0011	1	1111		MVN, MVNS (immediate) — MVNS	
0011	1	!= 1111		ORN, ORNS (immediate) — Flag setting	
0100	0			EOR, EORS (immediate) — EOR	
0100	1		1111	TEQ (immediate)	
0100	1		!= 1111	EOR, EORS (immediate) — EORS	
0101				UNALLOCATED	
011x				UNALLOCATED	
1000	0	1101		ADD, ADDS (SP plus immediate) — ADD	
1000	0	!= 1101		ADD, ADDS (immediate) — ADD	
1000	1		1111	CMN (immediate)	
1000	1	1101	!= 1111	ADD, ADDS (SP plus immediate) — ADDS	
1000	1	!= 1101	!= 1111	ADD, ADDS (immediate) — ADDS	
1001				UNALLOCATED	
1010	0			ADC, ADCS (immediate) — ADC	
1010	1			ADC, ADCS (immediate) — ADCS	
1011	0			SBC, SBCS (immediate) — SBC	
1011	1			SBC, SBCS (immediate) — SBCS	
1100				UNALLOCATED	
1101	0	1101		SUB, SUBS (SP minus immediate) — SUB	
1101	0	!= 1101		SUB, SUBS (immediate) — SUB	
1101	1		1111	CMP (immediate)	
1101	1	1101	!= 1111	SUB, SUBS (SP minus immediate) — SUBS	
1101	1	!= 1101	!= 1111	SUB, SUBS (immediate) — SUBS	
1110	0			RSB, RSBS (immediate) — RSB	
1110	1			RSB, RSBS (immediate) — RSBS	
1111				UNALLOCATED	

Data-processing (plain binary immediate)

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0		1	op0			op1	0				0																

Decode fields		Instruction details
op0	op1	
0	0x	Data-processing (simple immediate)
0	1 0	Move Wide (16-bit immediate)
0	1 1	UNALLOCATED
1		Saturate, Bitfield

Data-processing (simple immediate)

The encodings in this section are decoded from [Data-processing \(plain binary immediate\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	1	0	o1	0	o2	0	Rn				0	imm3				Rd				imm8							

Decode fields			Instruction Details
o1	o2	Rn	
0	0	1101	ADD, ADDS (SP plus immediate)
0	0	1111	ADR — T3
0	0	!= 11x1	ADD, ADDS (immediate)
0	1		UNALLOCATED
1	0		UNALLOCATED
1	1	1101	SUB, SUBS (SP minus immediate)
1	1	1111	ADR — T2
1	1	!= 11x1	SUB, SUBS (immediate)

Move Wide (16-bit immediate)

The encodings in this section are decoded from [Data-processing \(plain binary immediate\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	o1	1	0	0	imm4			0	imm3			Rd			imm8									

Decode fields	Instruction Details
o1	
0	MOV, MOVS (immediate)
1	MOVT

Saturate, Bitfield

The encodings in this section are decoded from [Data-processing \(plain binary immediate\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	op1			0	Rn			0	imm3			Rd			imm2			(0)	widthm1					

Decode fields			Instruction Details
op1	Rn	imm3:imm2	
000			SSAT — Logical shift left
001		00000	SSAT16
001		!= 00000	SSAT — Arithmetic shift right
010			SBFX
011	1111		BFC
011	!= 1111		BFI
100			USAT — Logical shift left
101		00000	USAT16
101		!= 00000	USAT — Arithmetic shift right
110			UBFX
111			UNALLOCATED

Branches and miscellaneous control

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
1	1	1	1	0	op0	op1					op2					1					op3					op4					op5									

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0	1110	0x	0x0		0	MSR (register)
0	1110	0x	0x0		1	MSR (Banked register)
0	1110	10	0x0	000		Hints
0	1110	10	0x0	!= 000		Change processor state
0	1110	11	0x0			Miscellaneous system
0	1111	00	0x0			BXJ
0	1111	01	0x0			Exception return
0	1111	1x	0x0		0	MRS
0	1111	1x	0x0		1	MRS (Banked register)
1	1110	00	000			DCPS
1	1110	00	010			UNALLOCATED
1	1110	!= 00	0x0			UNALLOCATED
1	1111	0x	0x0			UNALLOCATED
1	1111	1x	0x0			Exception generation
			0x1			B — T4
			1x0			BL, BLX (immediate) — T2
			1x1			BL, BLX (immediate) — T1
	!= 111x		0x0			B — T3

Hints

The encodings in this section are decoded from [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	0

Decode fields		Instruction Details	Condition
hint	option		
x101		Reserved hint, behaves as NOP	-
0x1x		Reserved hint, behaves as NOP	-
000x	1xxx	Reserved hint, behaves as NOP	-
0000	0000	NOP	-
0000	0001	YIELD	-
0000	0010	WFE	-
0000	0011	WFI	-
0000	0100	SEV	-
0000	0101	SEVL	-
0000	011x	Reserved hint, behaves as NOP	-
0001	0xx1	Reserved hint, behaves as NOP	-
0001	0000	ESB	FEAT_RAS
0001	0010	TSB	FEAT_TRF
0001	0100	CSDB	-
0001	0110	CLRBHB	FEAT_CLRBHB
0100		Reserved hint, behaves as NOP	-
10xx		Reserved hint, behaves as NOP	-
11x0		Reserved hint, behaves as NOP	-
1111		DBG	-

Change processor state

The encodings in this section are decoded from [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod	M	A	I	F	mode						

The following constraints also apply to this encoding: `!(imod == '00' && M == '0')`

Decode fields		Instruction Details
imod	M	
00	1	CPS, CPSID, CPSIE — Change mode
01		UNALLOCATED
10	0	CPS, CPSID, CPSIE — Interrupt enable
10	1	CPS, CPSID, CPSIE — Interrupt enable and change mode
11	0	CPS, CPSID, CPSIE — Interrupt disable
11	1	CPS, CPSID, CPSIE — Interrupt disable and change mode

Miscellaneous system

The encodings in this section are decoded from [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	opc			option				

Decode fields		Instruction Details		Condition
opc	option			
000x		UNALLOCATED		-
0010		CLREX		-
0011		UNALLOCATED		-
0100	0000	SSBB		-
0100	0100	PSSBB		-
0100	!= 0x00	DSB		-
0101		DMB		-
0110		ISB		-
0111		SB		FEAT_SB
1xxx		UNALLOCATED		-

Exception return

The encodings in this section are decoded from [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	1			Rn		1	0	(0)	0	(1)	(1)	(1)	(1)								imm8

Decode fields		Instruction Details	
Rn:imm8			
111000000000		ERET	
!= 111000000000		SUB, SUBS (immediate)	

DCPS

The encodings in this section are decoded from [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0			imm4		1	0	0	0												opt

Decode fields		Instruction Details	
imm4	imm10	opt	
1111	0000000000	00	UNALLOCATED
1111	0000000000	01	DCPS1
1111	0000000000	10	DCPS2
1111	0000000000	11	DCPS3
1111	!= 0000000000		UNALLOCATED
!= 1111			UNALLOCATED

Exception generation

The encodings in this section are decoded from [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	o1			imm4		1	0	o2	0												imm12

Decode fields		Instruction Details	
o1	o2		
0	0	HVC	
0	1	UNALLOCATED	
1	0	SMC	
1	1	UDF	

Advanced SIMD element or structure load/store

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	op0			0					op1															

Decode fields		Instruction details
op0	op1	
0		Advanced SIMD load/store multiple structures
1	11	Advanced SIMD load single structure to all lanes
1	!= 11	Advanced SIMD load/store single structure to one lane

Advanced SIMD load/store multiple structures

The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	L	0	Rn			Vd			itype			size		align		Rm						

L	Decode fields		Instruction Details
	itype	Rm	
	1011		UNALLOCATED
	11xx		UNALLOCATED
0	000x	1101	VST4 (multiple 4-element structures) — Post-indexed
0	000x	1111	VST4 (multiple 4-element structures) — Offset
0	000x	!= 11x1	VST4 (multiple 4-element structures) — Post-indexed
0	0010	1101	VST1 (multiple single elements) — T4 Post-indexed
0	0010	1111	VST1 (multiple single elements) — T4 Offset
0	0010	!= 11x1	VST1 (multiple single elements) — T4 Post-indexed
0	0011	1101	VST2 (multiple 2-element structures) — T2 Post-indexed
0	0011	1111	VST2 (multiple 2-element structures) — T2 Offset
0	0011	!= 11x1	VST2 (multiple 2-element structures) — T2 Post-indexed
0	010x	1101	VST3 (multiple 3-element structures) — Post-indexed
0	010x	1111	VST3 (multiple 3-element structures) — Offset
0	010x	!= 11x1	VST3 (multiple 3-element structures) — Post-indexed
0	0110	1101	VST1 (multiple single elements) — T3 Post-indexed
0	0110	1111	VST1 (multiple single elements) — T3 Offset
0	0110	!= 11x1	VST1 (multiple single elements) — T3 Post-indexed
0	0111	1101	VST1 (multiple single elements) — T1 Post-indexed
0	0111	1111	VST1 (multiple single elements) — T1 Offset
0	0111	!= 11x1	VST1 (multiple single elements) — T1 Post-indexed
0	100x	1101	VST2 (multiple 2-element structures) — T1 Post-indexed
0	100x	1111	VST2 (multiple 2-element structures) — T1 Offset
0	100x	!= 11x1	VST2 (multiple 2-element structures) — T1 Post-indexed
0	1010	1101	VST1 (multiple single elements) — T2 Post-indexed
0	1010	1111	VST1 (multiple single elements) — T2 Offset
0	1010	!= 11x1	VST1 (multiple single elements) — T2 Post-indexed
1	000x	1101	VLD4 (multiple 4-element structures) — Post-indexed
1	000x	1111	VLD4 (multiple 4-element structures) — Offset
1	000x	!= 11x1	VLD4 (multiple 4-element structures) — Post-indexed
1	0010	1101	VLD1 (multiple single elements) — T4 Post-indexed
1	0010	1111	VLD1 (multiple single elements) — T4 Offset
1	0010	!= 11x1	VLD1 (multiple single elements) — T4 Post-indexed
1	0011	1101	VLD2 (multiple 2-element structures) — T2 Post-indexed
1	0011	1111	VLD2 (multiple 2-element structures) — T2 Offset
1	0011	!= 11x1	VLD2 (multiple 2-element structures) — T2 Post-indexed
1	010x	1101	VLD3 (multiple 3-element structures) — Post-indexed
1	010x	1111	VLD3 (multiple 3-element structures) — Offset
1	010x	!= 11x1	VLD3 (multiple 3-element structures) — Post-indexed
1	0110	1101	VLD1 (multiple single elements) — T3 Post-indexed
1	0110	1111	VLD1 (multiple single elements) — T3 Offset
1	0110	!= 11x1	VLD1 (multiple single elements) — T3 Post-indexed
1	0111	1101	VLD1 (multiple single elements) — T1 Post-indexed
1	0111	1111	VLD1 (multiple single elements) — T1 Offset
1	0111	!= 11x1	VLD1 (multiple single elements) — T1 Post-indexed
1	100x	1101	VLD2 (multiple 2-element structures) — T1 Post-indexed

Decode fields			Instruction Details
L	itype	Rm	
1	100x	1111	VLD2 (multiple 2-element structures) — T1 Offset
1	100x	!= 11x1	VLD2 (multiple 2-element structures) — T1 Post-indexed
1	1010	1101	VLD1 (multiple single elements) — T2 Post-indexed
1	1010	1111	VLD1 (multiple single elements) — T2 Offset
1	1010	!= 11x1	VLD1 (multiple single elements) — T2 Post-indexed

Advanced SIMD load single structure to all lanes

The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	L	0	Rn				Vd				1	1	N	size	T	a	Rm					

Decode fields				Instruction Details
L	N	a	Rm	
0				UNALLOCATED
1	00		1101	VLD1 (single element to all lanes) — Post-indexed
1	00		1111	VLD1 (single element to all lanes) — Offset
1	00		!= 11x1	VLD1 (single element to all lanes) — Post-indexed
1	01		1101	VLD2 (single 2-element structure to all lanes) — Post-indexed
1	01		1111	VLD2 (single 2-element structure to all lanes) — Offset
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes) — Post-indexed
1	10	0	1101	VLD3 (single 3-element structure to all lanes) — Post-indexed
1	10	0	1111	VLD3 (single 3-element structure to all lanes) — Offset
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes) — Post-indexed
1	10	1		UNALLOCATED
1	11		1101	VLD4 (single 4-element structure to all lanes) — Post-indexed
1	11		1111	VLD4 (single 4-element structure to all lanes) — Offset
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes) — Post-indexed

Advanced SIMD load/store single structure to one lane

The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	L	0	Rn				Vd				!= 11	N	index	align	Rm							

size

The following constraints also apply to this encoding: size != '11'

Decode fields				Instruction Details
L	size	N	Rm	
0	00	00	1101	VST1 (single element from one lane) — T1 Post-indexed
0	00	00	1111	VST1 (single element from one lane) — T1 Offset
0	00	00	!= 11x1	VST1 (single element from one lane) — T1 Post-indexed
0	00	01	1101	VST2 (single 2-element structure from one lane) — T1 Post-indexed
0	00	01	1111	VST2 (single 2-element structure from one lane) — T1 Offset
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane) — T1 Post-indexed
0	00	10	1101	VST3 (single 3-element structure from one lane) — T1 Post-indexed
0	00	10	1111	VST3 (single 3-element structure from one lane) — T1 Offset
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane) — T1 Post-indexed
0	00	11	1101	VST4 (single 4-element structure from one lane) — T1 Post-indexed
0	00	11	1111	VST4 (single 4-element structure from one lane) — T1 Offset
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane) — T1 Post-indexed
0	01	00	1101	VST1 (single element from one lane) — T2 Post-indexed
0	01	00	1111	VST1 (single element from one lane) — T2 Offset
0	01	00	!= 11x1	VST1 (single element from one lane) — T2 Post-indexed
0	01	01	1101	VST2 (single 2-element structure from one lane) — T2 Post-indexed
0	01	01	1111	VST2 (single 2-element structure from one lane) — T2 Offset
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane) — T2 Post-indexed
0	01	10	1101	VST3 (single 3-element structure from one lane) — T2 Post-indexed
0	01	10	1111	VST3 (single 3-element structure from one lane) — T2 Offset
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane) — T2 Post-indexed
0	01	11	1101	VST4 (single 4-element structure from one lane) — T2 Post-indexed
0	01	11	1111	VST4 (single 4-element structure from one lane) — T2 Offset
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane) — T2 Post-indexed
0	10	00	1101	VST1 (single element from one lane) — T3 Post-indexed
0	10	00	1111	VST1 (single element from one lane) — T3 Offset
0	10	00	!= 11x1	VST1 (single element from one lane) — T3 Post-indexed
0	10	01	1101	VST2 (single 2-element structure from one lane) — T3 Post-indexed
0	10	01	1111	VST2 (single 2-element structure from one lane) — T3 Offset
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane) — T3 Post-indexed
0	10	10	1101	VST3 (single 3-element structure from one lane) — T3 Post-indexed
0	10	10	1111	VST3 (single 3-element structure from one lane) — T3 Offset
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane) — T3 Post-indexed
0	10	11	1101	VST4 (single 4-element structure from one lane) — T3 Post-indexed
0	10	11	1111	VST4 (single 4-element structure from one lane) — T3 Offset
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane) — T3 Post-indexed
1	00	00	1101	VLD1 (single element to one lane) — T1 Post-indexed
1	00	00	1111	VLD1 (single element to one lane) — T1 Offset
1	00	00	!= 11x1	VLD1 (single element to one lane) — T1 Post-indexed
1	00	01	1101	VLD2 (single 2-element structure to one lane) — T1 Post-indexed
1	00	01	1111	VLD2 (single 2-element structure to one lane) — T1 Offset
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane) — T1 Post-indexed
1	00	10	1101	VLD3 (single 3-element structure to one lane) — T1 Post-indexed
1	00	10	1111	VLD3 (single 3-element structure to one lane) — T1 Offset
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane) — T1 Post-indexed

Decode fields				Instruction Details
L	size	N	Rm	
1	00	11	1101	VLD4 (single 4-element structure to one lane) — T1 Post-indexed
1	00	11	1111	VLD4 (single 4-element structure to one lane) — T1 Offset
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane) — T1 Post-indexed
1	01	00	1101	VLD1 (single element to one lane) — T2 Post-indexed
1	01	00	1111	VLD1 (single element to one lane) — T2 Offset
1	01	00	!= 11x1	VLD1 (single element to one lane) — T2 Post-indexed
1	01	01	1101	VLD2 (single 2-element structure to one lane) — T2 Post-indexed
1	01	01	1111	VLD2 (single 2-element structure to one lane) — T2 Offset
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane) — T2 Post-indexed
1	01	10	1101	VLD3 (single 3-element structure to one lane) — T2 Post-indexed
1	01	10	1111	VLD3 (single 3-element structure to one lane) — T2 Offset
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane) — T2 Post-indexed
1	01	11	1101	VLD4 (single 4-element structure to one lane) — T2 Post-indexed
1	01	11	1111	VLD4 (single 4-element structure to one lane) — T2 Offset
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane) — T2 Post-indexed
1	10	00	1101	VLD1 (single element to one lane) — T3 Post-indexed
1	10	00	1111	VLD1 (single element to one lane) — T3 Offset
1	10	00	!= 11x1	VLD1 (single element to one lane) — T3 Post-indexed
1	10	01	1101	VLD2 (single 2-element structure to one lane) — T3 Post-indexed
1	10	01	1111	VLD2 (single 2-element structure to one lane) — T3 Offset
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane) — T3 Post-indexed
1	10	10	1101	VLD3 (single 3-element structure to one lane) — T3 Post-indexed
1	10	10	1111	VLD3 (single 3-element structure to one lane) — T3 Offset
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane) — T3 Post-indexed
1	10	11	1101	VLD4 (single 4-element structure to one lane) — T3 Post-indexed
1	10	11	1111	VLD4 (single 4-element structure to one lane) — T3 Offset
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane) — T3 Post-indexed

Load/store single

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	op0			op1			op2						op3												

The following constraints also apply to this encoding: !(op0 IN {'1x'}) && op1 == '0')

Decode fields				Instruction details
op0	op1	op2	op3	
00		!= 1111	000000	Load/store, unsigned (register offset)
00		!= 1111	000001	UNALLOCATED
00		!= 1111	00001x	UNALLOCATED
00		!= 1111	0001xx	UNALLOCATED
00		!= 1111	001xxx	UNALLOCATED
00		!= 1111	01xxxx	UNALLOCATED
00		!= 1111	10x0xx	UNALLOCATED
00		!= 1111	10x1xx	Load/store, unsigned (immediate, post-indexed)
00		!= 1111	1100xx	Load/store, unsigned (negative immediate)
00		!= 1111	1110xx	Load/store, unsigned (unprivileged)
00		!= 1111	11x1xx	Load/store, unsigned (immediate, pre-indexed)
01		!= 1111		Load/store, unsigned (positive immediate)
0x		1111		Load, unsigned (literal)
10	1	!= 1111	000000	Load/store, signed (register offset)
10	1	!= 1111	000001	UNALLOCATED
10	1	!= 1111	00001x	UNALLOCATED
10	1	!= 1111	0001xx	UNALLOCATED
10	1	!= 1111	001xxx	UNALLOCATED
10	1	!= 1111	01xxxx	UNALLOCATED
10	1	!= 1111	10x0xx	UNALLOCATED
10	1	!= 1111	10x1xx	Load/store, signed (immediate, post-indexed)
10	1	!= 1111	1100xx	Load/store, signed (negative immediate)
10	1	!= 1111	1110xx	Load/store, signed (unprivileged)
10	1	!= 1111	11x1xx	Load/store, signed (immediate, pre-indexed)
11	1	!= 1111		Load/store, signed (positive immediate)
1x	1	1111		Load, signed (literal)

Load/store, unsigned (register offset)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	1	1	0	0	0	0	size	L	!= 1111					Rt					0	0	0	0	0	0	imm2					Rm				
Rn																																				

The following constraints also apply to this encoding: Rn != '1111'

Decode fields			Instruction Details
size	L	Rt	
00	0		STRB (register)
00	1	1111	PLD, PLDW (register) — Preload read
00	1	!= 1111	LDRB (register)
01	0		STRH (register)
01	1	1111	PLD, PLDW (register) — Preload write
01	1	!= 1111	LDRH (register)
10	0		STR (register)
10	1		LDR (register)
11			UNALLOCATED

Load/store, unsigned (immediate, post-indexed)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	0	0	0	size	L	!= 1111					Rt					1	0	U	1	imm8										
Rn																P				W															

The following constraints also apply to this encoding: Rn != '1111'

Decode fields		Instruction Details	
size	L		
00	0	STRB (immediate)	
00	1	LDRB (immediate)	
01	0	STRH (immediate)	
01	1	LDRH (immediate)	
10	0	STR (immediate)	
10	1	LDR (immediate)	
11		UNALLOCATED	

Load/store, unsigned (negative immediate)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	1	1	1	0	0	0	0	size	L	!= 1111					Rt					1	1	0	0	imm8														
Rn																P				U				W															

The following constraints also apply to this encoding: Rn != '1111'

Decode fields			Instruction Details	
size	L	Rt		
00	0		STRB (immediate)	
00	1	1111	PLD, PLDW (immediate) — Preload read	
00	1	!= 1111	LDRB (immediate)	
01	0		STRH (immediate)	
01	1	1111	PLD, PLDW (immediate) — Preload write	
01	1	!= 1111	LDRH (immediate)	
10	0		STR (immediate)	
10	1		LDR (immediate)	
11			UNALLOCATED	

Load/store, unsigned (unprivileged)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	1	0	0	0	0	size	L	!= 1111					Rt					1	1	1	0	imm8									
Rn																																		

The following constraints also apply to this encoding: Rn != '1111'

Decode fields		Instruction Details
size	L	
00	0	STRBT
00	1	LDRBT
01	0	STRHT
01	1	LDRHT
10	0	STRT
10	1	LDRT
11		UNALLOCATED

Load/store, unsigned (immediate, pre-indexed)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	1	1	0	0	0	0	size	L		!= 1111				Rt				1	1	U	1							imm8					
												Rn								P		W													

The following constraints also apply to this encoding: Rn != '1111'

Decode fields		Instruction Details
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11		UNALLOCATED

Load/store, unsigned (positive immediate)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	size	L		!= 1111				Rt															imm12
												Rn																			

The following constraints also apply to this encoding: Rn != '1111'

Decode fields			Instruction Details
size	L	Rt	
00	0		STRB (immediate)
00	1	1111	PLD, PLDW (immediate) — Preload read
00	1	!= 1111	LDRB (immediate)
01	0		STRH (immediate)
01	1	1111	PLD, PLDW (immediate) — Preload write
01	1	!= 1111	LDRH (immediate)
10	0		STR (immediate)
10	1		LDR (immediate)
11			UNALLOCATED

Load, unsigned (literal)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	size	L	1	1	1	1		Rt															imm12

Decode fields										Instruction Details																			
size	L	Rt																											
	0											UNALLOCATED																	
0x	1	1111										PLD (literal)																	
00	1	!= 1111										LDRB (literal)																	
01	1	!= 1111										LDRH (literal)																	
10	1											LDR (literal)																	
11	1											UNALLOCATED																	

Load/store, signed (register offset)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	size	1		!= 1111					Rt				0	0	0	0	0	0	imm2		Rm		

Rn

The following constraints also apply to this encoding: Rn != '1111'

Decode fields										Instruction Details																			
size	Rt																												
00	1111										PLI (register)																		
00	!= 1111										LDRSB (register)																		
01	1111										Reserved hint, behaves as NOP																		
01	!= 1111										LDRSH (register)																		
1x											UNALLOCATED																		

Load/store, signed (immediate, post-indexed)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
1	1	1	1	1	0	0	1	0	size	1		!= 1111				Rt				1	0	U	1							imm8																	
Rn																P																W															

Rn

P

W

The following constraints also apply to this encoding: Rn != '1111'

Decode fields										Instruction Details																			
size																													
00											LDRSB (immediate)																		
01											LDRSH (immediate)																		
1x											UNALLOCATED																		

Load/store, signed (negative immediate)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	size	1		!= 1111				Rt				1	1	0	0							imm8	
Rn																P U W															

Rn

P

U

W

The following constraints also apply to this encoding: Rn != '1111'

Decode fields		Instruction Details
size	Rt	
00	1111	PLI (immediate, literal)
00	!= 1111	LDRSB (immediate)
01	1111	Reserved hint, behaves as NOP
01	!= 1111	LDRSH (immediate)
1x		UNALLOCATED

Load/store, signed (unprivileged)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	size	1	!= 1111		Rt	1	1	1	0		imm8												
Rn																															

The following constraints also apply to this encoding: Rn != '1111'

Decode fields		Instruction Details
size		
00		LDRSBT
01		LDRSHT
1x		UNALLOCATED

Load/store, signed (immediate, pre-indexed)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	size	1	!= 1111		Rt	1	1	U	1		imm8												
Rn																P		W													

The following constraints also apply to this encoding: Rn != '1111'

Decode fields		Instruction Details
size		
00		LDRSB (immediate)
01		LDRSH (immediate)
1x		UNALLOCATED

Load/store, signed (positive immediate)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	size	1	!= 1111		Rt																		
Rn																imm12															

The following constraints also apply to this encoding: Rn != '1111'

Decode fields		Instruction Details
size	Rt	
00	1111	PLI (immediate, literal)
00	!= 1111	LDRSB (immediate)
01	1111	Reserved hint, behaves as NOP
01	!= 1111	LDRSH (immediate)
1x		UNALLOCATED

Load, signed (literal)

The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	0	1	U	size	1	1	1	1	1		Rt					imm12											

Decode fields		Instruction Details
size	Rt	
00	1111	PLI (immediate, literal)
00	!= 1111	LDRSB (literal)
01	1111	Reserved hint, behaves as NOP
01	!= 1111	LDRSH (literal)
1x		UNALLOCATED

Data-processing (register)

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	op0								op1								op2							

Decode fields			Instruction details
op0	op1	op2	
0	1111	0000	Register shifts
0	1111	0001	UNALLOCATED
0	1111	001x	UNALLOCATED
0	1111	01xx	UNALLOCATED
0	1111	1xxx	Register extends
1	1111	0xxx	Parallel add-subtract
1	1111	10xx	Data-processing (two source registers)
1	1111	11xx	UNALLOCATED
	!= 1111		UNALLOCATED

Register shifts

The encodings in this section are decoded from [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	1	0	0	stype	S	Rm					1	1	1	1	Rd					0	0	0	0	Rs			

Decode fields		Instruction Details
S		
0		MOV, MOVS (register-shifted register) — Not flag setting
1		MOV, MOVS (register-shifted register) — Flag setting

Register extends

The encodings in this section are decoded from [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	op1	U	Rn					1	1	1	1	Rd					1 (0)	rotate	Rm				

Decode fields		Instruction Details	
op1	U	Rn	
00	0	1111	SXTH
00	0	!= 1111	SXTAH
00	1	1111	UXTH
00	1	!= 1111	UXTAH
01	0	1111	SXTB16
01	0	!= 1111	SXTAB16
01	1	1111	UXTB16
01	1	!= 1111	UXTAB16
10	0	1111	SXTB
10	0	!= 1111	SXTAB
10	1	1111	UXTB
10	1	!= 1111	UXTAB
11			UNALLOCATED

Parallel add-subtract

The encodings in this section are decoded from [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	1	0	1	op1				Rn				1	1	1	1	Rd				0	U	H	S	Rm			

Decode fields		Instruction Details		
op1	U	H	S	
x11				UNALLOCATED
!= x11		1	1	UNALLOCATED
000	0	0	0	SADD8
000	0	0	1	QADD8
000	0	1	0	SHADD8
000	1	0	0	UADD8
000	1	0	1	UQADD8
000	1	1	0	UHADD8
001	0	0	0	SADD16
001	0	0	1	QADD16
001	0	1	0	SHADD16
001	1	0	0	UADD16
001	1	0	1	UQADD16
001	1	1	0	UHADD16
010	0	0	0	SASX
010	0	0	1	QASX
010	0	1	0	SHASX
010	1	0	0	UASX
010	1	0	1	UQASX
010	1	1	0	UHASX
100	0	0	0	SSUB8
100	0	0	1	QSUB8
100	0	1	0	SHSUB8
100	1	0	0	USUB8
100	1	0	1	UQSUB8
100	1	1	0	UHSUB8
101	0	0	0	SSUB16
101	0	0	1	QSUB16
101	0	1	0	SHSUB16
101	1	0	0	USUB16
101	1	0	1	UQSUB16
101	1	1	0	UHSUB16
110	0	0	0	SSAX
110	0	0	1	QSAX
110	0	1	0	SHSAX
110	1	0	0	USAX
110	1	0	1	UQSAX
110	1	1	0	UHSAX

Data-processing (two source registers)

The encodings in this section are decoded from [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1		op1				Rn		1	1	1	1			Rd		1	0		op2			Rm	

Decode fields		Instruction Details	Condition
op1	op2		
000	00	QADD	-
000	01	QDADD	-
000	10	QSUB	-
000	11	QDSUB	-
001	00	REV	-
001	01	REV16	-
001	10	RBIT	-
001	11	REVSH	-
01x	!= 00	UNALLOCATED	-
010	00	SEL	-
011	00	CLZ	-
100	00	CRC32 — CRC32B	FEAT_CRC32
100	01	CRC32 — CRC32H	FEAT_CRC32
100	10	CRC32 — CRC32W	FEAT_CRC32
100	11	CONSTRAINED UNPREDICTABLE	-
101	00	CRC32C — CRC32CB	FEAT_CRC32
101	01	CRC32C — CRC32CH	FEAT_CRC32
101	10	CRC32C — CRC32CW	FEAT_CRC32
101	11	CONSTRAINED UNPREDICTABLE	-
11x		UNALLOCATED	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

Multiply, multiply accumulate, and absolute difference

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0																							

Decode fields		Instruction details
op0		
00		Multiply and absolute difference
!= 00		UNALLOCATED

Multiply and absolute difference

The encodings in this section are decoded from [Multiply, multiply accumulate, and absolute difference](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0																							

Decode fields		Instruction Details	
op1	Ra	op2	
000		01	MLS
000	1111	00	MUL, MULS
000	!= 1111	00	MLA, MLAS
001	1111	00	SMULBB, SMULBT, SMULTB, SMULTT — SMULBB
001	1111	01	SMULBB, SMULBT, SMULTB, SMULTT — SMULBT
001	1111	10	SMULBB, SMULBT, SMULTB, SMULTT — SMULTB
001	1111	11	SMULBB, SMULBT, SMULTB, SMULTT — SMULTT
001	!= 1111	00	SMLABB, SMLABT, SMLATB, SMLATT — SMLABB
001	!= 1111	01	SMLABB, SMLABT, SMLATB, SMLATT — SMLABT
001	!= 1111	10	SMLABB, SMLABT, SMLATB, SMLATT — SMLATB
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT — SMLATT
010	1111	00	SMUAD, SMUADX — SMUAD
010	1111	01	SMUAD, SMUADX — SMUADX
010	!= 1111	00	SMLAD, SMLADX — SMLAD
010	!= 1111	01	SMLAD, SMLADX — SMLADX
011	1111	00	SMULWB, SMULWT — SMULWB
011	1111	01	SMULWB, SMULWT — SMULWT
011	!= 1111	00	SMLAWB, SMLAWT — SMLAWB
011	!= 1111	01	SMLAWB, SMLAWT — SMLAWT
100	1111	00	SMUSD, SMUSDX — SMUSD
100	1111	01	SMUSD, SMUSDX — SMUSDX
100	!= 1111	00	SMLSD, SMLSDX — SMLSD
100	!= 1111	01	SMLSD, SMLSDX — SMLSDX
101	1111	00	SMMUL, SMMULR — SMMUL
101	1111	01	SMMUL, SMMULR — SMMULR
101	!= 1111	00	SMMLA, SMMLAR — SMMLA
101	!= 1111	01	SMMLA, SMMLAR — SMMLAR
110		00	SMMLS, SMMLSR — SMMLS
110		01	SMMLS, SMMLSR — SMMLSR
111		01	UNALLOCATED
111	1111	00	USAD8
111	!= 1111	00	USADA8
!= 001		1x	UNALLOCATED

Long multiply and divide

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	op1																						
Decode fields																Instruction details															
op1																															
0x1																Divide															
!= 0x1																Long Multiply															

Divide

The encodings in this section are decoded from [Long multiply and divide](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	0	x	1	Rn				Ra				Rd				op2				Rm			
op1																															

Decode fields

Instruction Details

op1	op2	
0x1	!= 1111	UNALLOCATED
001	1111	SDIV
011	1111	UDIV

Long Multiply

The encodings in this section are decoded from [Long multiply and divide](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	1	1	1	0	1	1	1	!= 0x1	Rn						RdLo						RdHi						op2						Rm					
op1																																							

The following constraints also apply to this encoding: !(op1 IN {'0x1'})

Decode fields

Instruction Details

op1	op2	
xx0	00x1	UNALLOCATED
xx0	0010	UNALLOCATED
0x0	!= 00xx	UNALLOCATED
000	0000	SMULL, SMULLS
010	0000	UMULL, UMULLS
1xx	0100	UNALLOCATED
1x0	01x1	UNALLOCATED
1x1	00xx	UNALLOCATED
1x1	0101	UNALLOCATED
1x1	011x	UNALLOCATED
100	x110	UNALLOCATED
100	0000	SMLAL, SMLALS
100	1000	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBB
100	1001	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBT
100	1010	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTB
100	1011	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTT
100	1100	SMLALD, SMLALDX — SMLALD
100	1101	SMLALD, SMLALDX — SMLALDX
100	1111	UNALLOCATED
101	10xx	UNALLOCATED
101	1100	SMLS LD, SMLS LDx — SMLS LD
101	1101	SMLS LD, SMLS LDx — SMLS LDx
101	111x	UNALLOCATED
11x	1xxx	UNALLOCATED
110	0000	UMLAL, UMLALS
110	0110	UMAAL

System register access, Advanced SIMD, and floating-point

The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op0	1	1	op1														op2							op3				

Decode fields				Instruction details
op0	op1	op2	op3	
0	0x	1x		Advanced SIMD and System register load/store and 64-bit move
0	10	10	0	Floating-point data-processing
0	10	11	0	UNALLOCATED
0	10	1x	1	Advanced SIMD and System register 32-bit move
1	!= 11	1x		Additional Advanced SIMD and floating-point instructions
	11			Advanced SIMD data-processing
	!= 11	0x		UNALLOCATED

Advanced SIMD and System register load/store and 64-bit move

The encodings in this section are decoded from [System register access, Advanced SIMD, and floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	1	0	1	1	0	op0																1	op1															

Decode fields		Instruction details
op0	op1	
00x0	0x	Advanced SIMD and floating-point 64-bit move
00x0	11	System register 64-bit move
	10	UNALLOCATED
!= 00x0	0x	Advanced SIMD and floating-point load/store
!= 00x0	11	System register Load/Store

Advanced SIMD and floating-point 64-bit move

The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	D	0	op	Rt2				Rt				1	0	size	opc2	M	o3	Vm					

Decode fields					Instruction Details
D	op	size	opc2	o3	
0					UNALLOCATED
1		0x			UNALLOCATED
1		1x	00	0	UNALLOCATED
1		1x	!= 00		UNALLOCATED
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — From general-purpose registers
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — From general-purpose registers
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — To general-purpose registers
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — To general-purpose registers

System register 64-bit move

The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	D	0	L	Rt2				Rt				1	1	1	cp15	opc1				CRm			

Decode fields		Instruction Details
D	L	
0		UNALLOCATED
1	0	MCRR
1	1	MRRC

Advanced SIMD and floating-point load/store

The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	L	Rn				Vd			1	0	size			imm8							

The following constraints also apply to this encoding: !(P == '0' && U == '0' && W == '0')

Decode fields						Instruction Details		Condition
P	U	W	L	Rn	size	imm8		
		1			0x		UNALLOCATED	-
0	0	1			1x		UNALLOCATED	-
0	1		0		10		VSTM, VSTMDB, VSTMIA — T2 Increment After	-
0	1		0		11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA — T1 Increment After	-
0	1		0		11	xxxxxxxx1	FSTMDBX, FSTMIAX — Increment After	-
0	1		1		10		VLDM, VLDMDB, VLDMIA — T2 Increment After	-
0	1		1		11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA — T1 Increment After	-
0	1		1		11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Increment After	-
0	1	0			0x		UNALLOCATED	-
1		0			00		UNALLOCATED	-
1		0	0		01		VSTR — Half-precision scalar	FEAT_FP16
1		0	0		10		VSTR — Single-precision scalar	-
1		0	0		11		VSTR — Double-precision scalar	-
1		0	1	1111	01		VLDR (literal) — Half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — Single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — Double-precision scalar	-
1		0	1	!= 1111	01		VLDR (immediate) — Half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — Single-precision scalar	-
1		0	1	!= 1111	11		VLDR (immediate) — Double-precision scalar	-
1	0	1	0		10		VSTM, VSTMDB, VSTMIA — T2 Decrement Before	-
1	0	1	0		11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA — T1 Decrement Before	-
1	0	1	0		11	xxxxxxxx1	FSTMDBX, FSTMIAX — Decrement Before	-
1	0	1	1		10		VLDM, VLDMDB, VLDMIA — T2 Decrement Before	-
1	0	1	1		11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA — T1 Decrement Before	-
1	0	1	1		11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before	-
1	1	1			1x		UNALLOCATED	-

System register Load/Store

The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	L	Rn			CRd			1	1	1	cp15	imm8									

The following constraints also apply to this encoding: !(P == '0' && U == '0' && W == '0')

P:U:W		Decode fields				Instruction Details	
	D	L	Rn	CRd	cp15		
0x1	0	0		0101	0	STC — Post-indexed	
0x1	0	1	!= 1111	0101	0	LDC (immediate) — Post-indexed	
010	0	0		0101	0	STC — Unindexed	
010	0	1	!= 1111	0101	0	LDC (immediate) — Unindexed	
1x0	0	0		0101	0	STC — Offset	
1x0	0	1	!= 1111	0101	0	LDC (immediate) — Offset	
1x1	0	0		0101	0	STC — Pre-indexed	
1x1	0	1	!= 1111	0101	0	LDC (immediate) — Pre-indexed	
!= 000	0			0101	1	UNALLOCATED	
!= 000	0			!= 0101		UNALLOCATED	
!= 000	0	1	1111	0101	0	LDC (literal)	
!= 000	1					UNALLOCATED	

Floating-point data-processing

The encodings in this section are decoded from [System register access, Advanced SIMD, and floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	op0								1	0	op1										0			

Decode fields		Instruction details	
op0	op1		
1x11	0	Floating-point move immediate	
1x11	1	Floating-point data-processing (two registers)	
!= 1x11		Floating-point data-processing (three registers)	

Floating-point move immediate

The encodings in this section are decoded from [Floating-point data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H				Vd				1	0	size		(0)	0	(0)	0	imm4L			

Decode fields		Instruction Details		Condition	
size					
00		UNALLOCATED		-	
01		VMOV (immediate) — Half-precision scalar		FEAT_FP16	
10		VMOV (immediate) — Single-precision scalar		-	
11		VMOV (immediate) — Double-precision scalar		-	

Floating-point data-processing (two registers)

The encodings in this section are decoded from [Floating-point data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	o1	opc2			Vd				1	0	size		o3	1	M	0	Vm			

Decode fields				Instruction Details	Condition
o1	opc2	size	o3		
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000	01	1	VABS — Half-precision scalar	FEAT_FP16
0	000	10	0	VMOV (register) — Single-precision scalar	-
0	000	10	1	VABS — Single-precision scalar	-
0	000	11	0	VMOV (register) — Double-precision scalar	-
0	000	11	1	VABS — Double-precision scalar	-
0	001	01	0	VNEG — Half-precision scalar	FEAT_FP16
0	001	01	1	VSQRT — Half-precision scalar	FEAT_FP16
0	001	10	0	VNEG — Single-precision scalar	-
0	001	10	1	VSQRT — Single-precision scalar	-
0	001	11	0	VNEG — Double-precision scalar	-
0	001	11	1	VSQRT — Double-precision scalar	-
0	010	01		UNALLOCATED	-
0	010	10	0	VCVTB — Half-precision to single-precision	-
0	010	10	1	VCVTT — Half-precision to single-precision	-
0	010	11	0	VCVTB — Half-precision to double-precision	-
0	010	11	1	VCVTT — Half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB — Single-precision to half-precision	-
0	011	10	1	VCVTT — Single-precision to half-precision	-
0	011	11	0	VCVTB — Double-precision to half-precision	-
0	011	11	1	VCVTT — Double-precision to half-precision	-
0	100	01	0	VCMP — T1 Half-precision scalar	FEAT_FP16
0	100	01	1	VCMPE — T1 Half-precision scalar	FEAT_FP16
0	100	10	0	VCMP — T1 Single-precision scalar	-
0	100	10	1	VCMPE — T1 Single-precision scalar	-
0	100	11	0	VCMP — T1 Double-precision scalar	-
0	100	11	1	VCMPE — T1 Double-precision scalar	-
0	101	01	0	VCMP — T2 Half-precision scalar	FEAT_FP16
0	101	01	1	VCMPE — T2 Half-precision scalar	FEAT_FP16
0	101	10	0	VCMP — T2 Single-precision scalar	-
0	101	10	1	VCMPE — T2 Single-precision scalar	-
0	101	11	0	VCMP — T2 Double-precision scalar	-
0	101	11	1	VCMPE — T2 Double-precision scalar	-
0	110	01	0	VRINTR — Half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — Half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — Single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — Single-precision scalar	-
0	110	11	0	VRINTR — Double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — Double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — Half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — Single-precision scalar	-

Decode fields				Instruction Details	Condition
o1	opc2	size	o3		
0	111	10	1	VCVT (between double-precision and single-precision) — Single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — Double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — Double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — Half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — Single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — Double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point) — Half-precision scalar	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point) — Single-precision scalar	-
1	01x	11		VCVT (between floating-point and fixed-point, floating-point) — Double-precision scalar	-
1	100	01	0	VCVTR — Half-precision scalar	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point) — Half-precision scalar	FEAT_FP16
1	100	10	0	VCVTR — Single-precision scalar	-
1	100	10	1	VCVT (floating-point to integer, floating-point) — Single-precision scalar	-
1	100	11	0	VCVTR — Double-precision scalar	-
1	100	11	1	VCVT (floating-point to integer, floating-point) — Double-precision scalar	-
1	101	01	0	VCVTR — Half-precision scalar	FEAT_FP16
1	101	01	1	VCVT (floating-point to integer, floating-point) — Half-precision scalar	FEAT_FP16
1	101	10	0	VCVTR — Single-precision scalar	-
1	101	10	1	VCVT (floating-point to integer, floating-point) — Single-precision scalar	-
1	101	11	0	VCVTR — Double-precision scalar	-
1	101	11	1	VCVT (floating-point to integer, floating-point) — Double-precision scalar	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point) — Half-precision scalar	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point) — Single-precision scalar	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point) — Double-precision scalar	-

Floating-point data-processing (three registers)

The encodings in this section are decoded from [Floating-point data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	o0	D	o1		Vn		Vd		1	0	size	N	o2	M	0									Vm

The following constraints also apply to this encoding: !(o0 == '1' && o1 == '11')

Decode fields				Instruction Details	Condition
o0	o1	size	o2		
0		00		UNALLOCATED	-
0	00	01	0	VMLA (floating-point) — Half-precision scalar	FEAT_FP16
0	00	01	1	VMLS (floating-point) — Half-precision scalar	FEAT_FP16
0	00	10	0	VMLA (floating-point) — Single-precision scalar	-
0	00	10	1	VMLS (floating-point) — Single-precision scalar	-
0	00	11	0	VMLA (floating-point) — Double-precision scalar	-
0	00	11	1	VMLS (floating-point) — Double-precision scalar	-
0	01	01	0	VNMLS — Half-precision scalar	FEAT_FP16
0	01	01	1	VNMLA — Half-precision scalar	FEAT_FP16
0	01	10	0	VNMLS — Single-precision scalar	-
0	01	10	1	VNMLA — Single-precision scalar	-
0	01	11	0	VNMLS — Double-precision scalar	-
0	01	11	1	VNMLA — Double-precision scalar	-
0	10	01	0	VMUL (floating-point) — Half-precision scalar	FEAT_FP16
0	10	01	1	VNMUL — Half-precision scalar	FEAT_FP16
0	10	10	0	VMUL (floating-point) — Single-precision scalar	-
0	10	10	1	VNMUL — Single-precision scalar	-
0	10	11	0	VMUL (floating-point) — Double-precision scalar	-
0	10	11	1	VNMUL — Double-precision scalar	-
0	11	01	0	VADD (floating-point) — Half-precision scalar	FEAT_FP16
0	11	01	1	VSUB (floating-point) — Half-precision scalar	FEAT_FP16
0	11	10	0	VADD (floating-point) — Single-precision scalar	-
0	11	10	1	VSUB (floating-point) — Single-precision scalar	-
0	11	11	0	VADD (floating-point) — Double-precision scalar	-
0	11	11	1	VSUB (floating-point) — Double-precision scalar	-
1	00	01	0	VDIV — Half-precision scalar	FEAT_FP16
1	00	10	0	VDIV — Single-precision scalar	-
1	00	11	0	VDIV — Double-precision scalar	-
1	00	!= 00	1	UNALLOCATED	-
1	01	01	0	VFNMS — Half-precision scalar	FEAT_FP16
1	01	01	1	VFNMA — Half-precision scalar	FEAT_FP16
1	01	10	0	VFNMS — Single-precision scalar	-
1	01	10	1	VFNMA — Single-precision scalar	-
1	01	11	0	VFNMS — Double-precision scalar	-
1	01	11	1	VFNMA — Double-precision scalar	-
1	10	01	0	VFMA — Half-precision scalar	FEAT_FP16
1	10	01	1	VFMS — Half-precision scalar	FEAT_FP16
1	10	10	0	VFMA — Single-precision scalar	-
1	10	10	1	VFMS — Single-precision scalar	-
1	10	11	0	VFMA — Double-precision scalar	-
1	10	11	1	VFMS — Double-precision scalar	-
1	!= 11	00		UNALLOCATED	-

Advanced SIMD and System register 32-bit move

The encodings in this section are decoded from [System register access](#), [Advanced SIMD](#), and [floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																		
1	1	1	0	1	1	1	0	op0																1	op1																1								

Decode fields		Instruction details
op0	op1	
000	000	UNALLOCATED
000	001	Floating-point 16-bit move
000	010	Floating-point 32-bit move
000	10x	UNALLOCATED
0x1	010	UNALLOCATED
10x	010	UNALLOCATED
111	010	Floating-point move special register
x10	010	UNALLOCATED
	011	Advanced SIMD 8/16/32-bit element move/duplicate
	11x	System register 32-bit move
!= 000	x0x	UNALLOCATED

Floating-point 16-bit move

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	0	op	Vn				Rt				1	0	0	1	N	(0)	(0)	1	(0)	(0)	(0)	(0)

Decode fields		Instruction Details	Condition
op			
0		VMOV (between general-purpose register and half-precision) — From general-purpose register	FEAT_FP16
1		VMOV (between general-purpose register and half-precision) — To general-purpose register	FEAT_FP16

Floating-point 32-bit move

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	0	op	Vn				Rt				1	0	1	0	N	(0)	(0)	1	(0)	(0)	(0)	(0)

Decode fields		Instruction Details
op		
0		VMOV (between general-purpose register and single-precision) — From general-purpose register
1		VMOV (between general-purpose register and single-precision) — To general-purpose register

Floating-point move special register

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	L	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

Decode fields		Instruction Details
L		
0		VMSR
1		VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1			L	Vn				Rt			1	0	1	1	N	opc2			1	(0)	(0)	(0)	(0)

Decode fields

opc1	L	opc2	Instruction Details
	1		VMOV (scalar to general-purpose register)
0xx	0		VMOV (general-purpose register to scalar)
1xx	0	1x	UNALLOCATED
1x0	0	0x	VDUP (general-purpose register) — 64-bit SIMD vector
1x1	0	0x	VDUP (general-purpose register) — 128-bit SIMD vector

System register 32-bit move

The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	1	1	0	opc1				L	CRn				Rt				1	1	1	cp15	opc2				1	CRm			

Decode fields

Instruction Details

L	
0	MCR
1	MRC

Additional Advanced SIMD and floating-point instructions

The encodings in this section are decoded from [System register access, Advanced SIMD, and floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	!= 11x		op1								1	op2	op3			op4			op5							
op0																															

The following constraints also apply to this encoding: !(op0 IN {'11x'})

Decode fields

Instruction details

op0	op1	op2	op3	op4	op5	
0xx			0x			Advanced SIMD three registers of the same length extension
0xx			1x			UNALLOCATED
100		0	!= 00	0	0	Floating-point conditional select
100		0	!= 00	1	0	UNALLOCATED
101	00xxxx	0	!= 00		0	Floating-point minNum/maxNum
101	01xxxx	0	!= 00		0	UNALLOCATED
101	10xxxx	0	!= 00		0	UNALLOCATED
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	110001	0	!= 00	1	0	UNALLOCATED
101	11001x	0	!= 00	1	0	UNALLOCATED
101	1101xx	0	!= 00	1	0	UNALLOCATED
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
101	11xxxx	0	!= 00	0	0	UNALLOCATED
10x		0	00			Advanced SIMD and floating-point multiply with accumulate
10x		0	!= 00		1	UNALLOCATED
10x		1	0x			Advanced SIMD and floating-point dot product
10x		1	1x			UNALLOCATED

Advanced SIMD three registers of the same length extension

The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	op1	D	op2	Vn			Vd			1	op3	0	op4	N	Q	M	U	Vm							

Decode fields						Instruction Details										Condition	
op1	op2	op3	op4	Q	U												
		0	1		0	UNALLOCATED										-	
	1x	0	0	0	0	VCMLA — 64-bit SIMD vector										FEAT_FCMA	
	1x	0	0	1	0	VCMLA — 128-bit SIMD vector										FEAT_FCMA	
x0	0x	0	0		0	UNALLOCATED										-	
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector										FEAT_FCMA	
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector										FEAT_FCMA	
0x	x1	1			0	UNALLOCATED										-	
0x	0x				1	UNALLOCATED										-	
0x	1x	0	1		1	UNALLOCATED										-	
0x	10	1	0	0	0	UNALLOCATED										-	
0x	11	1			1	UNALLOCATED										-	
00	00	1	0	0	0	UNALLOCATED										-	
00	00	1	0	1	0	VMMLA										FEAT_AA32BF16	
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector										FEAT_AA32BF16	
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector										FEAT_AA32BF16	
00	10	0	0	0	1	VFMAL (vector) — 64-bit SIMD vector										FEAT_FHM	
00	10	0	0	1	1	VFMAL (vector) — 128-bit SIMD vector										FEAT_FHM	
00	10	1	0	0	1	UNALLOCATED										-	
00	10	1	0	1	0	VSMMLA										FEAT_AA32I8MM	
00	10	1	0	1	1	YUMMLA										FEAT_AA32I8MM	
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector										FEAT_DotProd	
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector										FEAT_DotProd	
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector										FEAT_DotProd	
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector										FEAT_DotProd	
00	11	0	0		1	VFMAb, VFMAbT (BFloat16, vector)										FEAT_AA32BF16	
01	00	1			0	UNALLOCATED										-	
01	10	0	0	0	1	VFMSL (vector) — 64-bit SIMD vector										FEAT_FHM	
01	10	0	0	1	1	VFMSL (vector) — 128-bit SIMD vector										FEAT_FHM	
01	10	1			1	UNALLOCATED										-	
01	10	1	0	1	0	VUSMMLA										FEAT_AA32I8MM	
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector										FEAT_AA32I8MM	
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector										FEAT_AA32I8MM	
01	11	0	0		1	UNALLOCATED										-	
1x		0			1	UNALLOCATED										-	
1x		1				UNALLOCATED										-	

Floating-point conditional select

The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	cc		Vn		Vd		1	0	!= 00	N	0	M	0								Vm	
size																															

size

The following constraints also apply to this encoding: size != '00'

Decode fields		Instruction Details	Condition
cc	size		
00	01	VSELEQ, VSELGE, VSELGT, VSELVS — Equal, half-precision scalar	FEAT_FP16
00	10	VSELEQ, VSELGE, VSELGT, VSELVS — Equal, single-precision scalar	-
00	11	VSELEQ, VSELGE, VSELGT, VSELVS — Equal, double-precision scalar	-
01	01	VSELEQ, VSELGE, VSELGT, VSELVS — Unordered, half-precision scalar	FEAT_FP16
01	10	VSELEQ, VSELGE, VSELGT, VSELVS — Unordered, single-precision scalar	-
01	11	VSELEQ, VSELGE, VSELGT, VSELVS — Unordered, double-precision scalar	-
10	01	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than or Equal, half-precision scalar	FEAT_FP16
10	10	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than or Equal, single-precision scalar	-
10	11	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than or Equal, double-precision scalar	-
11	01	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than, half-precision scalar	FEAT_FP16
11	10	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than, single-precision scalar	-
11	11	VSELEQ, VSELGE, VSELGT, VSELVS — Greater than, double-precision scalar	-

Floating-point minNum/maxNum

The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn				Vd				1	0	!= 00		N	op	M	0	Vm			
																size															

The following constraints also apply to this encoding: size != '00'

Decode fields		Instruction Details	Condition
size	op		
01	0	VMAXNM — Half-precision scalar	FEAT_FP16
01	1	VMINNM — Half-precision scalar	FEAT_FP16
10	0	VMAXNM — Single-precision scalar	-
10	1	VMINNM — Single-precision scalar	-
11	0	VMAXNM — Double-precision scalar	-
11	1	VMINNM — Double-precision scalar	-

Floating-point extraction and insertion

The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd			1	0	!= 00		op	1	M	0	Vm				
																size															

The following constraints also apply to this encoding: size != '00'

Decode fields		Instruction Details	Condition
size	op		
×1		UNALLOCATED	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16

Floating-point directed convert to integer

The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	op1	RM			Vd			1	0	!= 00	op	1	M	0			Vm		
size																															

size

The following constraints also apply to this encoding: size != '00'

Decode fields				Instruction Details		Condition
op1	RM	size	op			
0		!= 00	1	UNALLOCATED		-
0	00	01	0	VRINTA (floating-point) — Half-precision scalar		FEAT_FP16
0	00	10	0	VRINTA (floating-point) — Single-precision scalar		-
0	00	11	0	VRINTA (floating-point) — Double-precision scalar		-
0	01	01	0	VRINTN (floating-point) — Half-precision scalar		FEAT_FP16
0	01	10	0	VRINTN (floating-point) — Single-precision scalar		-
0	01	11	0	VRINTN (floating-point) — Double-precision scalar		-
0	10	01	0	VRINTP (floating-point) — Half-precision scalar		FEAT_FP16
0	10	10	0	VRINTP (floating-point) — Single-precision scalar		-
0	10	11	0	VRINTP (floating-point) — Double-precision scalar		-
0	11	01	0	VRINTM (floating-point) — Half-precision scalar		FEAT_FP16
0	11	10	0	VRINTM (floating-point) — Single-precision scalar		-
0	11	11	0	VRINTM (floating-point) — Double-precision scalar		-
1	00	01		VCVTA (floating-point) — Half-precision scalar		FEAT_FP16
1	00	10		VCVTA (floating-point) — Single-precision scalar		-
1	00	11		VCVTA (floating-point) — Double-precision scalar		-
1	01	01		VCVTN (floating-point) — Half-precision scalar		FEAT_FP16
1	01	10		VCVTN (floating-point) — Single-precision scalar		-
1	01	11		VCVTN (floating-point) — Double-precision scalar		-
1	10	01		VCVTP (floating-point) — Half-precision scalar		FEAT_FP16
1	10	10		VCVTP (floating-point) — Single-precision scalar		-
1	10	11		VCVTP (floating-point) — Double-precision scalar		-
1	11	01		VCVTM (floating-point) — Half-precision scalar		FEAT_FP16
1	11	10		VCVTM (floating-point) — Single-precision scalar		-
1	11	11		VCVTM (floating-point) — Double-precision scalar		-

Advanced SIMD and floating-point multiply with accumulate

The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D		op2		Vn				Vd			1	0	0	0	N	Q	M	U		Vm		

Decode fields				Instruction Details	Condition
op1	op2	Q	U		
0		0	0	VCMLA (by element) — 64-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0		1	0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0	00	0	1	VFMAL (by scalar) — 64-bit SIMD vector	FEAT_FHM
0	00	1	1	VFMAL (by scalar) — 128-bit SIMD vector	FEAT_FHM
0	01	0	1	VFMSL (by scalar) — 64-bit SIMD vector	FEAT_FHM
0	01	1	1	VFMSL (by scalar) — 128-bit SIMD vector	FEAT_FHM
0	10		1	UNALLOCATED	-
0	11		1	VFMAb, VFMAb (BFloat16, by scalar)	FEAT_AA32BF16
1			1	UNALLOCATED	-
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point	FEAT_FCMA
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point	FEAT_FCMA

Advanced SIMD and floating-point dot product

The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	1	1	1	1	0	op1	D	op2	Vn					Vd					1	1	0	op4	N	Q	M	U	Vm				

Decode fields					Instruction Details	Condition
op1	op2	op4	Q	U		
	×0	0			UNALLOCATED	-
	×1				UNALLOCATED	-
0	00	1		1	UNALLOCATED	-
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector	FEAT_AA32BF16
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	FEAT_AA32BF16
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	10	1			UNALLOCATED	-

Advanced SIMD data-processing

The encodings in this section are decoded from [System register access, Advanced SIMD, and floating-point](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1		1	1	1	1	op0																	op1							

Decode fields		Instruction details
op0	op1	
0		Advanced SIMD three registers of the same length
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

Advanced SIMD three registers of the same length

The encodings in this section are decoded from [Advanced SIMD data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	U	1	1	1	1	0	D	size	Vn					Vd					opc					N	Q	M	o1	Vm				

U	Decode fields				Instruction Details	Condition
	size	opc	Q	o1		
		0000	0	0	VHADD — 64-bit SIMD vector	-
		0000	0	1	VQADD — 64-bit SIMD vector	-
		0000	1	0	VHADD — 128-bit SIMD vector	-
		0000	1	1	VQADD — 128-bit SIMD vector	-
		0001	0	0	VRHADD — 64-bit SIMD vector	-
		0001	1	0	VRHADD — 128-bit SIMD vector	-
		0010	0	0	VHSUB — 64-bit SIMD vector	-
		0010	0	1	VQSUB — 64-bit SIMD vector	-
		0010	1	0	VHSUB — 128-bit SIMD vector	-
		0010	1	1	VQSUB — 128-bit SIMD vector	-
		0011	0	0	VCGT (register) — T1 64-bit SIMD vector	-
		0011	0	1	VCGE (register) — T1 64-bit SIMD vector	-
		0011	1	0	VCGT (register) — T1 128-bit SIMD vector	-
		0011	1	1	VCGE (register) — T1 128-bit SIMD vector	-
		0100	0	0	VSHL (register) — 64-bit SIMD vector	-
		0100	0	1	VQSHL (register) — 64-bit SIMD vector	-
		0100	1	0	VSHL (register) — 128-bit SIMD vector	-
		0100	1	1	VQSHL (register) — 128-bit SIMD vector	-
		0101	0	0	VRSHL — 64-bit SIMD vector	-
		0101	0	1	VQRSHL — 64-bit SIMD vector	-
		0101	1	0	VRSHL — 128-bit SIMD vector	-
		0101	1	1	VQRSHL — 128-bit SIMD vector	-
		0110	0	0	VMAX (integer) — 64-bit SIMD vector	-
		0110	0	1	VMIN (integer) — 64-bit SIMD vector	-
		0110	1	0	VMAX (integer) — 128-bit SIMD vector	-
		0110	1	1	VMIN (integer) — 128-bit SIMD vector	-
		0111	0	0	VABD (integer) — 64-bit SIMD vector	-
		0111	0	1	VABA — 64-bit SIMD vector	-
		0111	1	0	VABD (integer) — 128-bit SIMD vector	-
		0111	1	1	VABA — 128-bit SIMD vector	-
		1001	0	1	VMUL (integer and polynomial) — 64-bit SIMD vector	-
		1001	1	1	VMUL (integer and polynomial) — 128-bit SIMD vector	-
		1010	0	0	VPMAX (integer)	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
0		1000	0	0	VADD (integer) — 64-bit SIMD vector	-
0		1000	0	1	VTST — 64-bit SIMD vector	-
0		1000	1	0	VADD (integer) — 128-bit SIMD vector	-
0		1000	1	1	VTST — 128-bit SIMD vector	-
0		1001	0	0	VMLA (integer) — 64-bit SIMD vector	-
0		1001	1	0	VMLA (integer) — 128-bit SIMD vector	-
0		1011		1	VPADD (integer)	-
0		1011	0	0	VQDMULH — 64-bit SIMD vector	-
0		1011	1	0	VQDMULH — 128-bit SIMD vector	-
0	0x	1100	0	1	VFMA — 64-bit SIMD vector	-

Decode fields					Instruction Details	Condition
U	size	opc	Q	o1		
0	0x	1100	1	1	VFMA — 128-bit SIMD vector	-
0	0x	1101	0	0	VADD (floating-point) — 64-bit SIMD vector	-
0	0x	1101	0	1	VMLA (floating-point) — 64-bit SIMD vector	-
0	0x	1101	1	0	VADD (floating-point) — 128-bit SIMD vector	-
0	0x	1101	1	1	VMLA (floating-point) — 128-bit SIMD vector	-
0	0x	1110		1	UNALLOCATED	-
0	0x	1110	0	0	VCEQ (register) — T2 64-bit SIMD vector	-
0	0x	1110	1	0	VCEQ (register) — T2 128-bit SIMD vector	-
0	0x	1111	0	0	VMAX (floating-point) — 64-bit SIMD vector	-
0	0x	1111	0	1	VRECPS — 64-bit SIMD vector	-
0	0x	1111	1	0	VMAX (floating-point) — 128-bit SIMD vector	-
0	0x	1111	1	1	VRECPS — 128-bit SIMD vector	-
0	00	0001	0	1	VAND (register) — 64-bit SIMD vector	-
0	00	0001	1	1	VAND (register) — 128-bit SIMD vector	-
0	00	1100		0	SHA1C	FEAT_SHA1
0	01	0001	0	1	VBIC (register) — 64-bit SIMD vector	-
0	01	0001	1	1	VBIC (register) — 128-bit SIMD vector	-
0	01	1100		0	SHA1P	FEAT_SHA1
0	1x	1100	0	1	VFMS — 64-bit SIMD vector	-
0	1x	1100	1	1	VFMS — 128-bit SIMD vector	-
0	1x	1101	0	0	VSUB (floating-point) — 64-bit SIMD vector	-
0	1x	1101	0	1	VMLS (floating-point) — 64-bit SIMD vector	-
0	1x	1101	1	0	VSUB (floating-point) — 128-bit SIMD vector	-
0	1x	1101	1	1	VMLS (floating-point) — 128-bit SIMD vector	-
0	1x	1110			UNALLOCATED	-
0	1x	1111	0	0	VMIN (floating-point) — 64-bit SIMD vector	-
0	1x	1111	0	1	VRSQRTS — 64-bit SIMD vector	-
0	1x	1111	1	0	VMIN (floating-point) — 128-bit SIMD vector	-
0	1x	1111	1	1	VRSQRTS — 128-bit SIMD vector	-
0	10	0001	0	1	VORR (register) — 64-bit SIMD vector	-
0	10	0001	1	1	VORR (register) — 128-bit SIMD vector	-
0	10	1100		0	SHA1M	FEAT_SHA1
0	11	0001	0	1	VORN (register) — 64-bit SIMD vector	-
0	11	0001	1	1	VORN (register) — 128-bit SIMD vector	-
0	11	1100		0	SHA1SU0	FEAT_SHA1
1		1000	0	0	VSUB (integer) — 64-bit SIMD vector	-
1		1000	0	1	VCEQ (register) — T1 64-bit SIMD vector	-
1		1000	1	0	VSUB (integer) — 128-bit SIMD vector	-
1		1000	1	1	VCEQ (register) — T1 128-bit SIMD vector	-
1		1001	0	0	VMLS (integer) — 64-bit SIMD vector	-
1		1001	1	0	VMLS (integer) — 128-bit SIMD vector	-
1		1011	0	0	VQRDMULH — 64-bit SIMD vector	-
1		1011	0	1	VQRDMULH — 64-bit SIMD vector	FEAT_RDM
1		1011	1	0	VQRDMULH — 128-bit SIMD vector	-
1		1011	1	1	VQRDMULH — 128-bit SIMD vector	FEAT_RDM
1		1100	0	1	VQRDMULSH — 64-bit SIMD vector	FEAT_RDM

Decode fields					Instruction Details	Condition
U	size	opc	Q	o1		
1		1100	1	1	VQRDMLSH — 128-bit SIMD vector	FEAT_RDM
1		1111	1	0	UNALLOCATED	-
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101	0	1	VMUL (floating-point) — 64-bit SIMD vector	-
1	0x	1101	1	1	VMUL (floating-point) — 128-bit SIMD vector	-
1	0x	1110	0	0	VCGE (register) — T2 64-bit SIMD vector	-
1	0x	1110	0	1	VACGE — 64-bit SIMD vector	-
1	0x	1110	1	0	VCGE (register) — T2 128-bit SIMD vector	-
1	0x	1110	1	1	VACGE — 128-bit SIMD vector	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111	0	1	VMAXNM — 64-bit SIMD vector	-
1	0x	1111	1	1	VMAXNM — 128-bit SIMD vector	-
1	00	0001	0	1	VEOR — 64-bit SIMD vector	-
1	00	0001	1	1	VEOR — 128-bit SIMD vector	-
1	00	1100		0	SHA256H	FEAT_SHA256
1	01	0001	0	1	VBSL — 64-bit SIMD vector	-
1	01	0001	1	1	VBSL — 128-bit SIMD vector	-
1	01	1100		0	SHA256H2	FEAT_SHA256
1	1x	1101		1	UNALLOCATED	-
1	1x	1101	0	0	VABD (floating-point) — 64-bit SIMD vector	-
1	1x	1101	1	0	VABD (floating-point) — 128-bit SIMD vector	-
1	1x	1110	0	0	VCGT (register) — T2 64-bit SIMD vector	-
1	1x	1110	0	1	VACGT — 64-bit SIMD vector	-
1	1x	1110	1	0	VCGT (register) — T2 128-bit SIMD vector	-
1	1x	1110	1	1	VACGT — 128-bit SIMD vector	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111	0	1	VMINNM — 64-bit SIMD vector	-
1	1x	1111	1	1	VMINNM — 128-bit SIMD vector	-
1	10	0001	0	1	VBIT — 64-bit SIMD vector	-
1	10	0001	1	1	VBIT — 128-bit SIMD vector	-
1	10	1100		0	SHA256SU1	FEAT_SHA256
1	11	0001	0	1	VBIF — 64-bit SIMD vector	-
1	11	0001	1	1	VBIF — 128-bit SIMD vector	-
1	11	1100		0	UNALLOCATED	-

Advanced SIMD two registers, or three registers of different lengths

The encodings in this section are decoded from [Advanced SIMD data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op0	1	1	1	1	1		op1						op2							op3			0					

Decode fields				Instruction details
op0	op1	op2	op3	
0	11			Advanced SIMD vector extract
1	11	0x		Advanced SIMD two registers misc
1	11	10		Advanced SIMD table permute
1	11	11		Advanced SIMD duplicate (scalar)
	!= 11		0	Advanced SIMD three registers of different lengths
	!= 11		1	Advanced SIMD two registers and a scalar

Advanced SIMD vector extract

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

Decode fields		Instruction Details
Q		
0		VEXT (byte elements) — 64-bit SIMD vector
1		VEXT (byte elements) — 128-bit SIMD vector

Advanced SIMD two registers misc

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size	opc1	Vd			0	opc2			Q	M	0	Vm							

Decode fields				Instruction Details	Condition
size	opc1	opc2	Q		
	00	0000	0	VREV64 — 64-bit SIMD vector	-
	00	0000	1	VREV64 — 128-bit SIMD vector	-
	00	0001	0	VREV32 — 64-bit SIMD vector	-
	00	0001	1	VREV32 — 128-bit SIMD vector	-
	00	0010	0	VREV16 — 64-bit SIMD vector	-
	00	0010	1	VREV16 — 128-bit SIMD vector	-
	00	0011		UNALLOCATED	-
	00	010x	0	VPADDL — 64-bit SIMD vector	-
	00	010x	1	VPADDL — 128-bit SIMD vector	-
	00	0110	0	AESE	FEAT_AES
	00	0110	1	AESD	FEAT_AES
	00	0111	0	AESMC	FEAT_AES
	00	0111	1	AESIMC	FEAT_AES
	00	1000	0	VCLS — 64-bit SIMD vector	-
	00	1000	1	VCLS — 128-bit SIMD vector	-
	00	1001	0	VCLZ — 64-bit SIMD vector	-
	00	1001	1	VCLZ — 128-bit SIMD vector	-
	00	1010	0	VCNT — 64-bit SIMD vector	-
	00	1010	1	VCNT — 128-bit SIMD vector	-
	00	1011	0	VMVN (register) — 64-bit SIMD vector	-
	00	1011	1	VMVN (register) — 128-bit SIMD vector	-
	00	110x	0	VPADAL — 64-bit SIMD vector	-
	00	110x	1	VPADAL — 128-bit SIMD vector	-
	00	1110	0	VQABS — 64-bit SIMD vector	-
	00	1110	1	VQABS — 128-bit SIMD vector	-
	00	1111	0	VQNEG — 64-bit SIMD vector	-
	00	1111	1	VQNEG — 128-bit SIMD vector	-
	01	x000	0	VCGT (immediate #0) — 64-bit SIMD vector	-
	01	x000	1	VCGT (immediate #0) — 128-bit SIMD vector	-
	01	x001	0	VCGE (immediate #0) — 64-bit SIMD vector	-
	01	x001	1	VCGE (immediate #0) — 128-bit SIMD vector	-
	01	x010	0	VCEQ (immediate #0) — 64-bit SIMD vector	-
	01	x010	1	VCEQ (immediate #0) — 128-bit SIMD vector	-
	01	x011	0	VCLE (immediate #0) — 64-bit SIMD vector	-
	01	x011	1	VCLE (immediate #0) — 128-bit SIMD vector	-
	01	x100	0	VCLT (immediate #0) — 64-bit SIMD vector	-
	01	x100	1	VCLT (immediate #0) — 128-bit SIMD vector	-
	01	x110	0	VABS — 64-bit SIMD vector	-
	01	x110	1	VABS — 128-bit SIMD vector	-
	01	x111	0	VNEG — 64-bit SIMD vector	-
	01	x111	1	VNEG — 128-bit SIMD vector	-
	01	0101	0	UNALLOCATED	-
	01	0101	1	SHA1H	FEAT_SHA1
	01	1101		UNALLOCATED	-
	10	x110	1	UNALLOCATED	-

size	Decode fields			Q	Instruction Details	Condition
	opc1	opc2				
	10	0001	0		VTRN — 64-bit SIMD vector	-
	10	0001	1		VTRN — 128-bit SIMD vector	-
	10	0010	0		VUZP — 64-bit SIMD vector	-
	10	0010	1		VUZP — 128-bit SIMD vector	-
	10	0011	0		VZIP — 64-bit SIMD vector	-
	10	0011	1		VZIP — 128-bit SIMD vector	-
	10	0100	0		VMOVN	-
	10	0100	1		VQMOVN, VQMOVUN — Unsigned result	-
	10	0101			VQMOVN, VQMOVUN — Signed result	-
	10	0110	0		VSHLL	-
	10	0111	0		SHA1SU1	FEAT_SHA1
	10	0111	1		SHA256SU0	FEAT_SHA256
	10	1000	0		VRINTN (Advanced SIMD) — 64-bit SIMD vector	-
	10	1000	1		VRINTN (Advanced SIMD) — 128-bit SIMD vector	-
	10	1001	0		VRINTX (Advanced SIMD) — 64-bit SIMD vector	-
	10	1001	1		VRINTX (Advanced SIMD) — 128-bit SIMD vector	-
	10	1010	0		VRINTA (Advanced SIMD) — 64-bit SIMD vector	-
	10	1010	1		VRINTA (Advanced SIMD) — 128-bit SIMD vector	-
	10	1011	0		VRINTZ (Advanced SIMD) — 64-bit SIMD vector	-
	10	1011	1		VRINTZ (Advanced SIMD) — 128-bit SIMD vector	-
	10	1100	0		VCVT (between half-precision and single-precision, Advanced SIMD) — Single-precision to half-precision	-
	10	1101	0		VRINTM (Advanced SIMD) — 64-bit SIMD vector	-
	10	1101	1		VRINTM (Advanced SIMD) — 128-bit SIMD vector	-
	10	1110	0		VCVT (between half-precision and single-precision, Advanced SIMD) — Half-precision to single-precision	-
	10	1111	0		VRINTP (Advanced SIMD) — 64-bit SIMD vector	-
	10	1111	1		VRINTP (Advanced SIMD) — 128-bit SIMD vector	-
	11	000x	0		VCVTA (Advanced SIMD) — 64-bit SIMD vector	-
	11	000x	1		VCVTA (Advanced SIMD) — 128-bit SIMD vector	-
	11	001x	0		VCVTN (Advanced SIMD) — 64-bit SIMD vector	-
	11	001x	1		VCVTN (Advanced SIMD) — 128-bit SIMD vector	-
	11	010x	0		VCVTP (Advanced SIMD) — 64-bit SIMD vector	-
	11	010x	1		VCVTP (Advanced SIMD) — 128-bit SIMD vector	-
	11	011x	0		VCVTM (Advanced SIMD) — 64-bit SIMD vector	-
	11	011x	1		VCVTM (Advanced SIMD) — 128-bit SIMD vector	-
	11	10x0	0		VRECPE — 64-bit SIMD vector	-
	11	10x0	1		VRECPE — 128-bit SIMD vector	-
	11	10x1	0		VRSQRTE — 64-bit SIMD vector	-
	11	10x1	1		VRSQRTE — 128-bit SIMD vector	-
	11	11xx	0		VCVT (between floating-point and integer, Advanced SIMD) — 64-bit SIMD vector	-
	11	11xx	1		VCVT (between floating-point and integer, Advanced SIMD) — 128-bit SIMD vector	-
00	10	0000	0		VSWP — 64-bit SIMD vector	-
00	10	0000	1		VSWP — 128-bit SIMD vector	-
01	10	1100	1		VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16

Decode fields				Instruction Details	Condition
size	opc1	opc2	Q		
!= 00	10	0000		UNALLOCATED	-
!= 01	10	1100	1	UNALLOCATED	-

Advanced SIMD table permute

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	D	1	1	Vn				Vd				1		0	len		N	op	M	0	Vm			

Decode fields		Instruction Details
op		
0		VTBL, VTBX — VTBL
1		VTBL, VTBX — VTBX

Advanced SIMD duplicate (scalar)

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	imm4				Vd				1		1	opc			Q	M	0	Vm		

Decode fields		Instruction Details
opc	Q	
000	0	VDUP (scalar)
000	1	VDUP (scalar)
!= 000		UNALLOCATED

Advanced SIMD three registers of different lengths

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	U	1	1	1	1	1	D	!= 11	Vn					Vd					opc					N	0	M	0	Vm				
size																																		

size

The following constraints also apply to this encoding: size != '11'

Decode fields		Instruction Details
U	opc	
	0000	VADDL
	0001	VADDW
	0010	VSUBL
	0011	VSUBW
	0101	VABAL
	0111	VABDL (integer)
	1000	VMLAL (integer)
	1010	VMLSL (integer)
	11×0	VMULL (integer and polynomial)
0	0100	VADDHN
0	0110	VSUBHN
0	1001	VQDMLAL
0	1011	VQDMLSL
0	1101	VQDMULL
0	1111	UNALLOCATED
1	0100	VRADDHN
1	0110	VRSUBHN
1	1××1	UNALLOCATED

Advanced SIMD two registers and a scalar

The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	Q	1	1	1	1	1	D	!= 11				Vn				Vd				opc				N	1	M	0	Vm			
size																																	

The following constraints also apply to this encoding: size != '11'

Decode fields		Instruction Details	Condition
Q	opc		
	0010	VMLAL (by scalar)	-
	0110	VMLSL (by scalar)	-
	1010	VMULL (by scalar)	-
0	000x	VMLA (by scalar) — 64-bit SIMD vector	-
0	0011	VQDMLAL	-
0	010x	VMLS (by scalar) — 64-bit SIMD vector	-
0	0111	VQDMLSL	-
0	100x	VMUL (by scalar) — 64-bit SIMD vector	-
0	1011	VQDMULL	-
0	1100	VQDMULH — 64-bit SIMD vector	-
0	1101	VQRDMULH — 64-bit SIMD vector	-
0	1110	VQRDMLAH — 64-bit SIMD vector	FEAT_RDM
0	1111	VQRDMLSH — 64-bit SIMD vector	FEAT_RDM
1	0x11	UNALLOCATED	-
1	000x	VMLA (by scalar) — 128-bit SIMD vector	-
1	010x	VMLS (by scalar) — 128-bit SIMD vector	-
1	100x	VMUL (by scalar) — 128-bit SIMD vector	-
1	1011	UNALLOCATED	-
1	1100	VQDMULH — 128-bit SIMD vector	-
1	1101	VQRDMULH — 128-bit SIMD vector	-
1	1110	VQRDMLAH — 128-bit SIMD vector	FEAT_RDM
1	1111	VQRDMLSH — 128-bit SIMD vector	FEAT_RDM

Advanced SIMD shifts and immediate generation

The encodings in this section are decoded from [Advanced SIMD data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1		1	1	1	1	1		op0																1					

Decode fields op0																Instruction details													
000xxxxxxxxxxx0																Advanced SIMD one register and modified immediate													
!= 000xxxxxxxxxxx0																Advanced SIMD two registers and shift amount													

Advanced SIMD one register and modified immediate

The encodings in this section are decoded from [Advanced SIMD shifts and immediate generation](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			cmode			0	Q	op	1	imm4					

Decode fields			Instruction Details
cmode	Q	op	
0xx0	0	0	VMOV (immediate) — T1 64-bit SIMD vector
0xx0	0	1	VMVN (immediate) — T1 64-bit SIMD vector
0xx0	1	0	VMOV (immediate) — T1 128-bit SIMD vector
0xx0	1	1	VMVN (immediate) — T1 128-bit SIMD vector
0xx1	0	0	VORR (immediate) — T1 64-bit SIMD vector
0xx1	0	1	VBIC (immediate) — T1 64-bit SIMD vector
0xx1	1	0	VORR (immediate) — T1 128-bit SIMD vector
0xx1	1	1	VBIC (immediate) — T1 128-bit SIMD vector
10x0	0	0	VMOV (immediate) — T3 64-bit SIMD vector
10x0	0	1	VMVN (immediate) — T2 64-bit SIMD vector
10x0	1	0	VMOV (immediate) — T3 128-bit SIMD vector
10x0	1	1	VMVN (immediate) — T2 128-bit SIMD vector
10x1	0	0	VORR (immediate) — T2 64-bit SIMD vector
10x1	0	1	VBIC (immediate) — T2 64-bit SIMD vector
10x1	1	0	VORR (immediate) — T2 128-bit SIMD vector
10x1	1	1	VBIC (immediate) — T2 128-bit SIMD vector
11xx	0	0	VMOV (immediate) — T4 64-bit SIMD vector
11xx	1	0	VMOV (immediate) — T4 128-bit SIMD vector
110x	0	1	VMVN (immediate) — T3 64-bit SIMD vector
110x	1	1	VMVN (immediate) — T3 128-bit SIMD vector
1110	0	1	VMOV (immediate) — T5 64-bit SIMD vector
1110	1	1	VMOV (immediate) — T5 128-bit SIMD vector
1111		1	UNALLOCATED

Advanced SIMD two registers and shift amount

The encodings in this section are decoded from [Advanced SIMD shifts and immediate generation](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm3H			imm3L			Vd			opc			L		Q	M	1	Vm				

The following constraints also apply to this encoding: !(imm3H == '000' && L == '0')

Decode fields						Instruction Details
U	imm3H	imm3L	opc	L	Q	
			1xxx	1		UNALLOCATED
	000		0000	1	0	VSHR — 64-bit SIMD vector
	000		0000	1	1	VSHR — 128-bit SIMD vector
	000		0001	1	0	VSRA — 64-bit SIMD vector
	000		0001	1	1	VSRA — 128-bit SIMD vector
	000		0010	1	0	VRSHR — 64-bit SIMD vector
	000		0010	1	1	VRSHR — 128-bit SIMD vector
	000		0011	1	0	VRSRA — 64-bit SIMD vector
	000		0011	1	1	VRSRA — 128-bit SIMD vector
	000		0111	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
	000		0111	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
	001	000	1010	0	0	VMOVL
	001	!= 000	1010	0	0	VSHLL
	010	000	1010	0	0	VMOVL
	010	!= 000	1010	0	0	VSHLL
	011		1010	0	0	VSHLL
	100	000	1010	0	0	VMOVL
	100	!= 000	1010	0	0	VSHLL
	101		1010	0	0	VSHLL
	11x		1010	0	0	VSHLL
	!= 000		0000		0	VSHR — 64-bit SIMD vector
	!= 000		0000		1	VSHR — 128-bit SIMD vector
	!= 000		0001		0	VSRA — 64-bit SIMD vector
	!= 000		0001		1	VSRA — 128-bit SIMD vector
	!= 000		0010		0	VRSHR — 64-bit SIMD vector
	!= 000		0010		1	VRSHR — 128-bit SIMD vector
	!= 000		0011		0	VRSRA — 64-bit SIMD vector
	!= 000		0011		1	VRSRA — 128-bit SIMD vector
	!= 000		0111		0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
	!= 000		0111		1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
	!= 000		1001	0	0	VQSHRN, VQSHRUN — Signed result
	!= 000		1001	0	1	VQRSHRN, VQRSHRUN — Signed result

Decode fields						Instruction Details
U	imm3H	imm3L	opc	L	Q	
	!= 000		1010	0	1	UNALLOCATED
	!= 000		1011	0		UNALLOCATED
	!= 000		11xx	0	0	VCVT (between floating-point and fixed-point, Advanced SIMD) — 64-bit SIMD vector
	!= 000		11xx	0	1	VCVT (between floating-point and fixed-point, Advanced SIMD) — 128-bit SIMD vector
0			01x0	1		UNALLOCATED
0	000		0101	1	0	VSHL (immediate) — 64-bit SIMD vector
0	000		0101	1	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000		01x0	0		UNALLOCATED
0	!= 000		0101		0	VSHL (immediate) — 64-bit SIMD vector
0	!= 000		0101		1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000		1000	0	0	VSHRN
0	!= 000		1000	0	1	VRSHRN
1	000		0100	1	0	VSRI — 64-bit SIMD vector
1	000		0100	1	1	VSRI — 128-bit SIMD vector
1	000		0101	1	0	VSLI — 64-bit SIMD vector
1	000		0101	1	1	VSLI — 128-bit SIMD vector
1	000		0110	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	000		0110	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000		0100		0	VSRI — 64-bit SIMD vector
1	!= 000		0100		1	VSRI — 128-bit SIMD vector
1	!= 000		0101		0	VSLI — 64-bit SIMD vector
1	!= 000		0101		1	VSLI — 128-bit SIMD vector
1	!= 000		0110		0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	!= 000		0110		1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000		1000	0	0	VQSHRN, VQSHRUN — Unsigned result
1	!= 000		1000	0	1	VQRSHRN, VQRSHRUN — Unsigned result

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved.

Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages

Pseudocode

Library pseudocode for aarch32/at/AArch32_AT

```
// AArch32_AT()
// =====
// Perform address translation as per AT instructions.

func AArch32_AT(vaddress : bits(32), stage_in : TranslationStage, el : bits(2), ataccess : ATAccess)
begin
    var stage : TranslationStage = stage_in;
    var ss : SecurityState;
    var regime : Regime;
    var eae : boolean;

    // ATS1Hx instructions
    if el == EL2 then
        regime = Regime\_EL2;
        eae = TRUE;
        ss = SS\_NonSecure;

    // ATS1Cxx instructions
    elsif stage == TranslationStage\_1 || (stage == TranslationStage\_12 && !HaveEL(EL2)) then
        stage = TranslationStage\_1;
        ss = SecurityStateAtEL(PSTATE.EL);
        regime = if ss == SS\_Secure && ELUsingAArch32(EL3) then Regime\_EL30 else Regime\_EL10;
        eae = TTBCR().EAE == '1';

    // ATS12NSOxx instructions
    else
        regime = Regime\_EL10;
        eae = (if HaveEL(EL3) && ELUsingAArch32(EL3) then TTBCR\_NS().EAE == '1'
              else TTBCR().EAE == '1');
        ss = SS\_NonSecure;
    end;

    var addrdesc : AddressDescriptor;
    var sdftype : SDFTYPE;
    let aligned : boolean = TRUE;
    var supersection : bit = '0';

    let accdesc : AccessDescriptor = CreateAccDescAT(ss, el, ataccess);

    // Prepare fault fields in case a fault is detected
    var fault : FaultRecord = NoFault(accdesc, ZeroExtend(64){vaddress});

    if eae then
        (fault, addrdesc) = AArch32\_S1TranslateLD(fault, regime, vaddress, aligned, accdesc);
    else
        (fault, addrdesc, sdftype) = AArch32\_S1TranslateSD(fault, regime, vaddress, aligned, accdesc);
        supersection = if sdftype == SDFTYPE\_Supersection then '1' else '0';
    end;

    // ATS12NSOxx instructions
    if stage == TranslationStage\_12 && fault.statuscode == Fault\_None then
        (fault, addrdesc) = AArch32\_S2Translate(fault, addrdesc, aligned, accdesc);
    end;

    if fault.statuscode != Fault\_None then
        // Take exception on External abort or when a fault occurs on translation table walk
        if IsExternalAbort(fault) || (PSTATE.EL == EL1 && EL2Enabled() && fault.s2fslwalk) then
            PAR() = ARBITRARY : bits(64);
            AArch32\_Abort(fault);
        end;
    end;

    addrdesc.fault = fault;

    if (eae || (stage == TranslationStage\_12 && (HCR().VM == '1' || HCR().DC == '1'))
        || (stage == TranslationStage\_1 && el != EL2 && PSTATE.EL == EL2)) then
        AArch32\_EncodePARLD(addrdesc, ss);
    else
        AArch32\_EncodePARSD(addrdesc, supersection, ss);
    end;
    return;
end;
```


Library pseudocode for aarch32/at/AArch32_EncodePARLD

```
// AArch32_EncodePARLD()
// =====
// Returns 64-bit format PAR on address translation instruction.

func AArch32_EncodePARLD(addrdesc : AddressDescriptor, ss : SecurityState)
begin
    PAR()[63:0] = Zeros{64};
    if !IsFault(addrdesc) then
        var ns : bit;
        if ss == SS_NonSecure then
            ns = ARBITRARY : bit;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        end;
        PAR().F = '0';
        PAR().SH = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR().NS = ns;
        PAR()[10] = ImpDefBit("Non-Faulting PAR"); // IMPDEF
        PAR().LPAE = '1';
        PAR().PA = addrdesc.paddress.address[39:12];
        PAR().ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
    else
        PAR().F = '1';
        PAR().FST = AArch32_PARFaultStatusLD(addrdesc.fault);
        PAR().S2WLK = if addrdesc.fault.s2fslwalk then '1' else '0';
        PAR().FSTAGE = if addrdesc.fault.secondstage then '1' else '0';
        PAR().LPAE = '1';
        PAR()[63:48] = ImpDefBits{16}("Faulting PAR"); // IMPDEF
    end;
    return;
end;
```

Library pseudocode for aarch32/at/AArch32_EncodePARSD

```
// AArch32_EncodePARSD()
// =====
// Returns 32-bit format PAR on address translation instruction.

func AArch32_EncodePARSD(addrdesc_in : AddressDescriptor, supersection : bit, ss : SecurityState)
begin
    PAR()[31:0] = Zeros{32};
    var addrdesc : AddressDescriptor = addrdesc_in;
    if !IsFault(addrdesc) then
        if (addrdesc.memattrs.memtype == MemType_Device ||
            (addrdesc.memattrs.inner.attrs == MemAttr_NC &&
             addrdesc.memattrs.outer.attrs == MemAttr_NC)) then
            addrdesc.memattrs.shareability = Shareability_OSH;
        end;
        var ns : bit;
        if ss == SS_NonSecure then
            ns = ARBITRARY : bit;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        end;
        let sh : bits(2) = (if addrdesc.memattrs.shareability != Shareability_NSH then '01'
                           else '00');
        PAR().F = '0';
        PAR().SS = supersection;
        PAR().Outer = AArch32_ReportedOuterAttrs(AArch32_PAROuterAttrs(addrdesc.memattrs));
        PAR().Inner = AArch32_ReportedInnerAttrs(AArch32_PARInnerAttrs(addrdesc.memattrs));
        PAR().SH = ReportedPARShareability(sh);
        PAR()[8] = ImpDefBit("Non-Faulting PAR"); // IMPDEF
        PAR().NS = ns;
        PAR().NOS = if addrdesc.memattrs.shareability == Shareability_OSH then '0' else '1';
        PAR().LPAE = '0';
        PAR().PA = addrdesc.paddress.address[39:12];
    else
        PAR().F = '1';
        PAR().FST = AArch32_PARFaultStatusSD(addrdesc.fault);
        PAR().LPAE = '0';
        PAR()[31:16] = ImpDefBits{16}("Faulting PAR"); // IMPDEF
    end;
    return;
end;
```

Library pseudocode for aarch32/at/AArch32_PARFaultStatusLD

```
// AArch32_PARFaultStatusLD()
// =====
// Fault status field decoding of 64-bit PAR

func AArch32_PARFaultStatusLD(fault : FaultRecord) => bits(6)
begin
    var syndrome : bits(6);
    if fault.statuscode == Fault\_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        syndrome[1:0] = if fault.level == 1 then '01' else '10';
        syndrome[5:2] = '1111';
    else
        syndrome = EncodeLDFSC(fault.statuscode, fault.level);
    end;
    return syndrome;
end;
```

Library pseudocode for aarch32/at/AArch32_PARFaultStatusSD

```
// AArch32_PARFaultStatusSD()
// =====
// Fault status field decoding of 32-bit PAR.

func AArch32_PARFaultStatusSD(fault : FaultRecord) => bits(6)
begin
    var syndrome : bits(6);

    syndrome[5] = if IsExternalAbort(fault) then fault.extflag else '0';
    syndrome[4:0] = EncodeSDFSC(fault.statuscode, fault.level);
    return syndrome;
end;
```

Library pseudocode for aarch32/at/AArch32_PARInnerAttrs

```
// AArch32_PARInnerAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Inner field.

func AArch32_PARInnerAttrs(memattrs : MemoryAttributes) => bits(3)
begin
    var result : bits(3);

    if memattrs.memtype == MemType\_Device then
        if memattrs.device == DeviceType\_nGnRnE then
            result = '001'; // Non-cacheable
        elsif memattrs.device == DeviceType\_nGnRE then
            result = '011'; // Non-cacheable
        end;
    else
        let inner : MemAttrHints = memattrs.inner;
        if inner.attrs == MemAttr\_NC then
            result = '000'; // Non-cacheable
        elsif inner.attrs == MemAttr\_WB && inner.hints[0] == '1' then
            result = '101'; // Write-Back, Write-Allocate
        elsif inner.attrs == MemAttr\_WT then
            result = '110'; // Write-Through
        elsif inner.attrs == MemAttr\_WB && inner.hints[0] == '0' then
            result = '111'; // Write-Back, no Write-Allocate
        end;
    end;
    return result;
end;
```

Library pseudocode for aarch32/at/AArch32_PAROuterAttrs

```
// AArch32_PAROuterAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Outer field.

func AArch32_PAROuterAttrs(memattrs : MemoryAttributes) => bits(2)
begin
    var result : bits(2);

    if memattrs.memtype == MemType\_Device then
        result = ARBITRARY : bits(2);
    else
        let outer : MemAttrHints = memattrs.outer;
        if outer.attrs == MemAttr\_NC then
            result = '00'; // Non-cacheable
        elsif outer.attrs == MemAttr\_WB && outer.hints[0] == '1' then
            result = '01'; // Write-Back, Write-Allocate
        elsif outer.attrs == MemAttr\_WT && outer.hints[0] == '0' then
            result = '10'; // Write-Through, no Write-Allocate
        elsif outer.attrs == MemAttr\_WB && outer.hints[0] == '0' then
            result = '11'; // Write-Back, no Write-Allocate
        end;
    end;
    return result;
end;
```

Library pseudocode for aarch32/at/AArch32_ReportedInnerAttrs

```
// AArch32_ReportedInnerAttrs()
// =====
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

impdef func AArch32_ReportedInnerAttrs(attrs : bits(3)) => bits(3)
begin
    return attrs;
end;
```

Library pseudocode for aarch32/at/AArch32_ReportedOuterAttrs

```
// AArch32_ReportedOuterAttrs()
// =====
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

impdef func AArch32_ReportedOuterAttrs(attrs : bits(2)) => bits(2)
begin
    return attrs;
end;
```

Library pseudocode for aarch32/dc/AArch32_CanTrapDC

```
// AArch32_CanTrapDC()
// =====
// Determines whether the execution of the DC instruction can be trapped.

func AArch32_CanTrapDC(cacheop : CacheOp, opscope : CacheOpScope) => boolean
begin
    return (!AArch32\_TreatDCAsNOP(cacheop, opscope) || ImpDefBool(
        "When DC is treated as NOP, data cache maintenance operations are trapped"));
end;
```

Library pseudocode for aarch32/dc/AArch32_DC

```
// AArch32_DC()
// =====
// Perform Data Cache Operation.

func AArch32_DC(regval : bits(32), cacheop : CacheOp, opscope : CacheOpScope)
begin
    var cache : CacheRecord;

    cache.acctype = AccessType\_DC;
    cache.cacheop = cacheop;
    cache.opscope = opscope;
    cache.cachetype = CacheType\_Data;
    cache.security = SecurityStateAtEL(PSTATE.EL);

    if opscope == CacheOpScope\_SetWay then
        cache.shareability = Shareability\_NSH;
        cache.(setnum, waynum, level) = DecodeSW(ZeroExtend{64}(regval),
                                                CacheType\_Data);

        if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            ((ELUsingAArch32(EL2) && (HCR_EL2().SWIO == '1' || HCR_EL2().[DC,VM] != '00')) ||
             (ELUsingAArch32(EL2) && (HCR().SWIO == '1' || HCR().[DC,VM] != '00')))) then
            cache.cacheop = CacheOp\_CleanInvalidate;
        end;
        CACHE_OP(cache);
        return;
    end;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid = VMID();
        else
            cache.is_vmid_valid = FALSE;
        end;
    else
        cache.is_vmid_valid = FALSE;
    end;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid = ASID();
    else
        cache.is_asid_valid = FALSE;
    end;

    var vaddress : bits(32) = regval;

    var size : integer = 0; // by default no watchpoint address
    if cacheop == CacheOp\_Invalidate then
        size = DataCacheWatchpointSize();
        vaddress = AlignDownSize{32}(regval, size as AddressSize);
    end;

    cache.vaddress = ZeroExtend{64}(vaddress);

    let aligned : boolean = TRUE;
    let accdesc : AccessDescriptor = CreateAccDescDC(cache);
    var memaddrdesc : AddressDescriptor = AArch32\_TranslateAddress(vaddress, accdesc,
                                                                    aligned, size);

    if IsFault(memaddrdesc) then
        memaddrdesc.fault.vaddress = ZeroExtend{64}(regval);
        AArch32\_Abort(memaddrdesc.fault);
    end;

    cache.paddress = memaddrdesc.paddress;
    if opscope == CacheOpScope\_PoC then
        cache.shareability = memaddrdesc.memattr.shareability;
    else
        cache.shareability = Shareability\_NSH;
    end;

    if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
        ((ELUsingAArch32(EL2) && HCR_EL2().[DC,VM] != '00') ||
         (ELUsingAArch32(EL2) && HCR().[DC,VM] != '00')))) then
        cache.cacheop = CacheOp\_CleanInvalidate;
    end;

    CACHE_OP(cache);
    return;
end;
```

Library pseudocode for aarch32/dc/AArch32_TreatDCAsNOP

```
// AArch32_TreatDCAsNOP()
// =====
// Determines whether the execution of the DC instruction is treated as a NOP.

func AArch32_TreatDCAsNOP(cacheop : CacheOp, opscope : CacheOpScope) => boolean
begin
    // DC to PoU: IMPLEMENTATION DEFINED - treated as NOP if LoUU and LoUIS are 0
    if opscope == CacheOpScope\_PoU && CLIDR().LoUU == '000' && CLIDR().LoUIS == '000' then
        return ImpDefBool("DC to PoU is treated as a NOP");
    end;

    // DC to PoC: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0
    if opscope == CacheOpScope\_PoC && CLIDR().LoC == '000' then
        return ImpDefBool("DC to PoC is treated as a NOP");
    end;

    return FALSE;
end;
```

Library pseudocode for aarch32/debug/VCRMatch/AArch32_VCRMatch

```
// AArch32_VCRMatch()
// =====

func AArch32_VCRMatch(vaddress : bits(32)) => boolean
begin
    var match : boolean;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        var match_word : bits(32) = Zeros{};

        let ss : SecurityState = CurrentSecurityState();
        if vaddress[31:5] == ExcVectorBase() [31:5] then
            if HaveEL(EL3) && ss == SS\_NonSecure then
                match_word[UInt(vaddress[4:2]) + 24] = '1'; // Non-secure vectors
            else
                match_word[UInt(vaddress[4:2]) + 0] = '1'; // Secure vectors (or no EL3)
            end;
        end;
        if (HaveEL(EL3) && ELUsingAArch32(EL3) && vaddress[31:5] == MVBAR() [31:5] &&
            ss == SS\_Secure) then
            match_word[UInt(vaddress[4:2]) + 8] = '1'; // Monitor vectors
        end;
        // Mask out bits not corresponding to vectors.
        var mask : bits(32);
        if !HaveEL(EL3) then
            mask = '00000000'::'00000000'::'00000000'::'11011110'; // DBGVCR[31:8] are RES0
        elseif !ELUsingAArch32(EL3) then
            mask = '11011110'::'00000000'::'00000000'::'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110'::'00000000'::'11011100'::'11011110';
        end;

        match_word = match_word AND DBGVCR() AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word[28:27,12:11,4:3]) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHDAPA);
        end;

        if !IsZero(vaddress[1:0]) && match then
            match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHHALF);
        end;
    else
        match = FALSE;
    end;

    return match;
end;
```

Library pseudocode for aarch32/debug/authentication/AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// =====

func AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled() => boolean
begin
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && NonSecureOnlyImplementation() then return FALSE; end;
    return DBGEN == HIGH && SPIDEN == HIGH;
end;
```

Library pseudocode for aarch32/debug/breakpoint/AArch32_BreakpointMatch

```
// AArch32_BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.

func AArch32_BreakpointMatch(n : integer, vaddress : bits(32), accdesc : AccessDescriptor,
                             size : integer) => BreakpointInfo
begin
    assert ELUsingAArch32(S1TranslationRegime() );
    assert n < NumBreakpointsImplemented();

    var brkptinfo : BreakpointInfo;
    let enabled : boolean = DBGBCR(n).E == '1';
    let isbreakpnt : boolean = TRUE;
    let linked : boolean = DBGBCR(n).BT == '0x01';
    let linked_to : boolean = FALSE;
    let linked_n : integer{} = UInt(DBGBCR(n).LBN);

    let state_match : boolean = AArch32\_StateMatch(DBGBCR(n).SSC, DBGBCR(n).HMC, DBGBCR(n).PMC,
                                                linked, linked_n, isbreakpnt, accdesc);
    var (value_match, value_mismatch) = AArch32\_BreakpointValueMatch(n,vaddress, linked_to);

    if size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether this breakpoint generates a Breakpoint debug
        // event.
        let (match_i, mismatch_i) = AArch32\_BreakpointValueMatch(n, vaddress + 2, linked_to);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);
        end;

        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable\_BPMISMATCHHALF);
        end;
    end;

    if vaddress[1] == '1' && DBGBCR(n).BAS == '1111' then
        // The above notwithstanding, if DBGBCR(n).BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR(n)+2.
        if value_match then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);
        end;

        if !value_mismatch then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable\_BPMISMATCHHALF);
        end;
    end;

    brkptinfo.match = value_match && state_match && enabled;
    brkptinfo.mismatch = value_mismatch && state_match && enabled;

    return brkptinfo;
end;
```



```

// AArch32_BreakpointValueMatch()
// =====
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

func AArch32_BreakpointValueMatch(n_in : integer, vaddress : bits(32),
                                  linked_to : boolean) => (boolean, boolean)
begin
    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    // matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    var n : integer = n_in;
    var c : Constraint;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                              Unpredictable BPNOTIMPL);
        assert c IN {Constraint DISABLED, Constraint UNKNOWN};
        if c == Constraint DISABLED then return (FALSE, FALSE); end;
    end;

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if DBGBCR(n).E == '0' then return (FALSE, FALSE); end;

    var dbgtype : bits(4) = DBGBCR(n).BT;

    (c, dbgtype) = AArch32\_ReservedBreakpointType(n, dbgtype);
    if c == Constraint DISABLED then return (FALSE, FALSE); end;
    // Otherwise the dbgtype value returned by AArch32_ReservedBreakpointType is valid.

    // Determine what to compare against.
    let match_addr : boolean = (dbgtype == '0x0x');
    let mismatch : boolean = (dbgtype == '010x');
    let match_vmid : boolean = (dbgtype == '10xx');
    let match_cid1 : boolean = (dbgtype == 'xx1x');
    let match_cid2 : boolean = (dbgtype == '11xx');
    let linking_enabled : boolean = (dbgtype == 'xxx1');

    // If called from StateMatch, is CONSTRAINED UNPREDICTABLE if the
    // breakpoint is not programmed with linking enabled.
    if linked_to && !linking_enabled then
        if !ConstrainUnpredictableBool(Unpredictable BPLINKINGDISABLED) then
            return (FALSE, FALSE);
        end;
    end;

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linking_enabled && !match_addr then
        return (FALSE, FALSE);
    end;

    var bvr_match : boolean = FALSE;
    var bxvr_match : boolean = FALSE;

    // Do the comparison.
    if match_addr then
        let byte : integer = UInt(vaddress[1:0]);
        assert byte IN {0,2}; // "vaddress" is halfword aligned

        let byte_select_match : boolean = (DBGBCR(n).BAS[byte] == '1');
        bvr_match = (vaddress[31:2] == DBGBCR(n)[31:2]) && byte_select_match;
    elsif match_cid1 then
        bvr_match = (PSTATE.EL != EL2 && CONTEXTIDR() == DBGBCR(n)[31:0]);
    end;

    if match_vmid then
        let vmid : bits(NUM_VMIDBITS) = VMID();
        var bvr_vmid : bits(NUM_VMIDBITS);

        if ELUsingAArch32(EL2) then
            bvr_vmid = ZeroExtend{NUM_VMIDBITS}(DBGBCR(n)[7:0]);
        elsif !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2().VS == '0' then
            bvr_vmid = ZeroExtend{NUM_VMIDBITS}(DBGBCR(n)[7:0]);
        else
            bvr_vmid = DBGBCR(n)[15:0];
        end;

        bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && vmid == bvr_vmid);
    elsif match_cid2 then
        bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() && !ELUsingAArch32(EL2) &&
                     DBGBCR(n)[31:0] == CONTEXTIDR\_EL2() [31:0]);
    end;
end

```



```

end;

let bvr_match_valid : boolean = (match_addr || match_cid1);
let bxvr_match_valid : boolean = (match_vmid || match_cid2);

let match : boolean = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

return (match && !mismatch, !match && mismatch);
end;

```

Library pseudocode for aarch32/debug/breakpoint/AArch32_ReservedBreakpointType

```

// AArch32_ReservedBreakpointType()
// =====
// Checks if the given DBGBCR(n).BT value is reserved and will generate Constrained Unpredictable
// behavior, otherwise returns Constraint_NONE.

func AArch32_ReservedBreakpointType(n : integer, bt_in : bits(4)) => (Constraint, bits(4))
begin
    var bt : bits(4) = bt_in;
    var reserved : boolean = FALSE;
    let context_aware : boolean = IsContextAwareBreakpoint(n);

    // Address mismatch
    if bt == '010x' && HaltOnBreakpointOrWatchpoint() then
        reserved = TRUE;
    end;

    // Context matching
    if bt != '0x0x' && !context_aware then
        reserved = TRUE;
    end;

    // EL2 extension
    if bt == '1xxx' && !HaveEL(EL2) then
        reserved = TRUE;
    end;

    // Context matching
    if (bt IN {'011x', '11xx'}) && !IsFeatureImplemented(FEAT_VHE) &&
        !IsFeatureImplemented(FEAT_Debugv8p2)) then
        reserved = TRUE;
    end;

    if reserved then
        var c : Constraint;
        (c, bt) = ConstrainUnpredictableBits{4}(Unpredictable_RESBPTYPE);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, ARBITRARY : bits(4));
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
        end;
    end;

    return (Constraint_NONE, bt);
end;

```

Library pseudocode for aarch32/debug/breakpoint/AArch32_StateMatch

```
// AArch32_StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

func AArch32_StateMatch(ssc_in : bits(2), hmc_in : bit, pxc_in : bits(2), linked_in : boolean,
    linked_n_in : integer, isbreakpnt : boolean,
    accdesc : AccessDescriptor) => boolean

begin

    // "ssc_in","hmc_in","pxc_in" are the control fields from the DBGBCR(n) or DBGWCR(n) register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR(n) or DBGWCR(n) register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "accdesc" describes the properties of the access being matched.
    var hmc : bit = hmc_in;
    var ssc : bits(2) = ssc_in;
    var pxc : bits(2) = pxc_in;
    var linked : boolean = linked_in;
    var linked_n : integer = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    var c : Constraint;
    // SSCE value discarded as there is no SSCE bit in AArch32_
    (c, ssc, -, hmc, pxc) = CheckValidStateMatch(ssc, '0', hmc, pxc, isbreakpnt);
    if c == Constraint DISABLED then return FALSE; end;
    // Otherwise the hmc,ssc,pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    let pl2_match : boolean = HaveEL(EL2) && ((hmc == '1' && (ssc::pxc != '1000')) || ssc == '11');
    let pl1_match : boolean = pxc[0] == '1';
    let pl0_match : boolean = pxc[1] == '1';
    let ssu_match : boolean = isbreakpnt && hmc == '0' && pxc == '00' && ssc != '11';

    var priv_match : boolean;
    if ssu_match then
        priv_match = PSTATE.M IN {M32\_User,M32\_Svc,M32\_System};
    else
        case accdesc.el of
            when EL3 => priv_match = pl1_match; // EL3 and EL1 are both PL1
            when EL2 => priv_match = pl2_match;
            when EL1 => priv_match = pl1_match;
            when EL0 => priv_match = pl0_match;
        end;
    end;

    // Security state match
    var ss_match : boolean;
    case ssc of
        when '00' => ss_match = TRUE; // Both
        when '01' => ss_match = accdesc.ss == SS\_NonSecure; // Non-secure only
        when '10' => ss_match = accdesc.ss == SS\_Secure; // Secure only
        when '11' => ss_match = (hmc == '1' || accdesc.ss == SS\_Secure); // HMC=1 -> Both,
        // HMC=0 -> Secure only
    end;

    var linked_match : boolean = FALSE;

    if linked then
        // "linked_n" must be an enabled context-aware breakpoint unit.
        // If it is not context-aware then it is CONSTRAINED UNPREDICTABLE whether
        // this gives no match, gives a match without linking, or linked_n is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        if IsContextAwareBreakpoint(linked_n) then
            let (first_ctx_cmp, last_ctx_cmp) : (integer, integer) = ContextAwareBreakpointRange();
            (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
                Unpredictable BPNOTCTXCMP);
            assert c IN {Constraint DISABLED, Constraint NONE, Constraint UNKNOWN};

            case c of
                when Constraint DISABLED => return FALSE; // Disabled
                when Constraint NONE => linked = FALSE; // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
            end;
        end;

        let vaddress : bits(32) = ARBITRARY : bits(32);
        let linked_to : boolean = TRUE;
        (linked_match,-) = AArch32\_BreakpointValueMatch(linked_n, vaddress, linked_to);
    end;

    return priv_match && ss_match && (!linked || linked_match);
end;
```

Library pseudocode for aarch32/debug/enables/AArch32_GenerateDebugExceptions

```
// AArch32_GenerateDebugExceptions()
// =====

func AArch32_GenerateDebugExceptions() => boolean
begin
    let ss : SecurityState = CurrentSecurityState();
    return AArch32\_GenerateDebugExceptionsFrom(PSTATE.EL, ss);
end;
```

Library pseudocode for aarch32/debug/enables/AArch32_GenerateDebugExceptionsFrom

```
// AArch32_GenerateDebugExceptionsFrom()
// =====

func AArch32_GenerateDebugExceptionsFrom(from_el : bits(2), from_state : SecurityState) => boolean
begin
    if !ELUsingAArch32(DebugTargetFrom(from_state)) then
        let mask : bit = '0'; // No PSTATE.D in AArch32 state
        return AArch64\_GenerateDebugExceptionsFrom(from_el, from_state, mask);
    end;
    if DBGOSLSR().OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    end;

    var enabled : boolean;
    if HaveEL(EL3) && from_state == SS\_Secure then
        assert from_el != EL2; // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR\_EL3().SDD == '0';
        else
            let spd : bits(2) = if ELUsingAArch32(EL3) then SDCR().SPD else MDCR\_EL3().SPD32;
            if spd[1] == '1' then
                enabled = spd[0] == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b00.
                enabled = AArch32\_SelfHostedSecurePrivilegedInvasiveDebugEnabled();
            end;
        end;
        if from_el == EL0 then enabled = enabled || SDER().SUIDEN == '1'; end;
    else
        enabled = from_el != EL2;
    end;

    return enabled;
end;
```

Library pseudocode for aarch32/debug/pmu/AArch32_IncrementCycleCounter

```
// AArch32_IncrementCycleCounter()
// =====
// Increment the cycle counter and possibly set overflow bits.

func AArch32_IncrementCycleCounter()
begin
    if !CountPMUEvents(CYCLE\_COUNTER\_ID) then return; end;
    var d : bit = PMCR().D; // Check divide-by-64
    var lc : bit = PMCR().LC;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0'; end;
    if d == '1' && !HasElapsed64Cycles() then return; end;

    let old_value : integer = UInt(PMCCNTR());
    let new_value : integer = old_value + 1;
    PMCCNTR() = new_value[63:0];

    let ovflw : integer{} = if lc == '1' then 64 else 32;

    if old_value[64:ovflw] != new_value[64:ovflw] then
        PMOVSSET().C = '1';
    end;

    return;
end;
```

Library pseudocode for aarch32/debug/pmu/AArch32_IncrementEventCounter

```
// AArch32_IncrementEventCounter()
// =====
// Increment the specified event counter 'idx' by the specified amount 'increment'.
// 'Vm' is the value event counter 'idx-1' is being incremented by if 'idx' is odd,
// zero otherwise.
// Returns the amount the counter was incremented by.

func AArch32_IncrementEventCounter(idx : integer, increment_in : integer, Vm : integer) => integer
begin
    if HaveAArch64() then
        // Force the counter to be incremented as a 64-bit counter.
        return AArch64_IncrementEventCounter(idx, increment_in, Vm);
    end;

    // In this model, event counters in an AArch32-only implementation are 32 bits and
    // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is implemented.
    var old_value : integer;
    var new_value : integer;

    old_value = UInt(PMEVCNTR(idx));
    let increment : integer = PMUCountValue(idx, increment_in, Vm);
    new_value = old_value + increment;

    PMEVCNTR(idx) = new_value[31:0];
    let overflow : integer{} = 32;
    if old_value[64:overflow] != new_value[64:overflow] then
        PMOVSSET()[idx] = '1';
        // Check for the CHAIN event from an even counter
        if (idx[0] == '0' && idx + 1 < NUM_PMU_COUNTERS &&
            (GetPMUCounterRange(idx) == GetPMUCounterRange(idx+1) ||
             ConstrainUnpredictableBool(Unpredictable_COUNT_CHAIN))) then
            // If PMU counters idx and idx+1 are not in same range,
            // it is CONSTRAINED UNPREDICTABLE if CHAIN event is counted
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
        end;
    end;

    return increment;
end;
```

Library pseudocode for aarch32/debug/pmu/AArch32_PMUCycle

```
// AArch32_PMUCycle()
// =====
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

func AArch32_PMUCycle()
begin
    if HaveAArch64() then
        AArch64_PMUCycle();
        return;
    end;

    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    end;

    PMUEvent(PMU_EVENT_CPU_CYCLES);
    let counters : integer = NUM_PMU_COUNTERS;
    var Vm : integer = 0;
    if counters != 0 then
        for idx = 0 to counters - 1 do
            if CountPMUEvents(idx) then
                let accumulated : integer = PMUEventAccumulator[[idx]];
                if (idx MOD 2) == 0 then Vm = 0; end;
                Vm = AArch32_IncrementEventCounter(idx, accumulated, Vm);
            end;
            PMUEventAccumulator[[idx]] = 0;
        end;
    end;

    AArch32_IncrementCycleCounter();
    CheckForPMUOverflow();
end;
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32_EnterHypModeInDebugState

```
// AArch32_EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

noreturn func AArch32_EnterHypModeInDebugState(except : ExceptionRecord)
begin
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS\_NonSecure && ELUsingAArch32(EL2);

    AArch32\_ReportHypEntry(except);
    AArch32\_WriteMode(M32\_Hyp);
    SPSR\_curr() = ARBITRARY : bits(32);
    ELR\_hyp() = ARBITRARY : bits(32);
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.[SS,A,I,F] are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.\[SS,A,I,F\] = ARBITRARY : bits(4);
    DLR() = ARBITRARY : bits(32);
    DPSPR() = ARBITRARY : bits(32);
    if IsFeatureImplemented(FEAT\_Debugv8p9) then
        DPSPR2() = ARBITRARY : bits(32);
    end;
    PSTATE.E = HSCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT\_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
    EDSCR().ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction(EndOfInstructionReason\_Exception);
end;
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32_EnterModeInDebugState

```
// AArch32_EnterModeInDebugState()
// =====
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

noreturn func AArch32_EnterModeInDebugState(target_mode : bits(5))
begin
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32\_Monitor then SCR().NS = '0'; end;
    AArch32\_WriteMode(target_mode);
    SPSR\_curr() = ARBITRARY : bits(32);
    R(14) = ARBITRARY : bits(32);
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.[SS,A,I,F] are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.\[SS,A,I,F\] = ARBITRARY : bits(4);
    DLR() = ARBITRARY : bits(32);
    DPSPR() = ARBITRARY : bits(32);
    if IsFeatureImplemented(FEAT\_Debugv8p9) then
        DPSPR2() = ARBITRARY : bits(32);
    end;
    PSTATE.E = SCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT\_PAN) && SCTLR().SPAN == '0' then PSTATE.PAN = '1'; end;
    if IsFeatureImplemented(FEAT\_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
    EDSCR().ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR PE state flags.

    EndOfInstruction(EndOfInstructionReason\_Exception);
end;
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32_EnterMonitorModeInDebugState

```
// AArch32_EnterMonitorModeInDebugState()
// =====
// Take an exception in Debug state to Monitor mode.

func AArch32_EnterMonitorModeInDebugState()
begin
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    let from_secure : boolean = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
    AArch32_WriteMode(M32_Monitor);
    SPSR_curr() = ARBITRARY : bits(32);
    R(14) = ARBITRARY : bits(32);
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.[SS,A,I,F] are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.[SS,A,I,F] = ARBITRARY : bits(4);
    PSTATE.E = SCTLRL().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elseif SCTLRL().SPAN == '0' then
            PSTATE.PAN = '1';
        end;
    end;
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2() = ARBITRARY : bits(32);
    end;
    EDSCR().ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR PE state flags.

    EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

Library pseudocode for aarch32/debug/watchpoint/AArch32_WatchpointByteMatch

```
// AArch32_WatchpointByteMatch()
// =====

func AArch32_WatchpointByteMatch(n : integer, vaddress : bits(32)) => boolean
begin
    let dbgtop : integer{} = 31;
    let cmpbottom : integer{} = if DBGWVR(n)[2] == '1' then 2 else 3; // Word or doubleword
    var bottom = cmpbottom;
    let select : integer{} = UInt(vaddress[cmpbottom-1:0]);
    var byte_select_match : boolean = (DBGWCR(n).BAS[select] != '0');
    var mask : integer{} = UInt(DBGWCR(n).MASK);

    // If DBGWCR(n).MASK is a nonzero value and DBGWCR(n).BAS is not set to '11111111', or
    // DBGWCR(n).BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR(n).BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable WPMASKANDBAS);
    else
        let LSB : bits(8) = (DBGWCR(n).BAS AND NOT(DBGWCR(n).BAS - 1));
        let MSB : bits(8) = (DBGWCR(n).BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable WPBASCONTIGUOUS);
            bottom = 3; // For the whole doubleword
        end;
    end;

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        var c : Constraint;
        var unpred_mask : integer;
        (c, unpred_mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable RESWPMASK);
        assert c IN {Constraint DISABLED, Constraint NONE, Constraint UNKNOWN};
        case c of
            when Constraint DISABLED => return FALSE; // Disabled
            when Constraint NONE => mask = 0; // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value
            otherwise => mask = unpred_mask as integer{3..31};
        end;
    end;

    let cmpmsb : integer{} = dbgtop;
    let cmpls : integer{} = if mask > bottom then mask else bottom;
    let bottombit : integer{} = bottom;
    var WVR_match : boolean = (vaddress[cmpmsb:cmpls] == DBGWVR(n)[cmpmsb:cmpls]);
    if mask > bottom then
        // If masked bits of DBGWVR(n) are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR(n)[cmpls-1:bottombit]) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable WPMASKEDBITS);
        end;
    end;

    return (WVR_match && byte_select_match);
end;
```

Library pseudocode for aarch32/debug/watchpoint/AArch32_WatchpointMatch

```
// AArch32_WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

func AArch32_WatchpointMatch(n : integer, vaddress : bits(32), size : integer,
    accdesc : AccessDescriptor) => WatchpointInfo
begin
    assert ELUsingAArch32(S1TranslationRegime()) ;
    assert n < NumWatchpointsImplemented() ;

    let enabled : boolean      = DBGWCR(n).E == '1';
    let linked : boolean      = DBGWCR(n).WT == '1';
    let isbreakpnt : boolean  = FALSE;
    let linked_n : integer{}  = UInt(DBGWCR_EL1(n).LBN);
    let state_match : boolean = AArch32\_StateMatch(DBGWCR(n).SSC, DBGWCR(n).HMC, DBGWCR(n).PAC,
        linked, linked_n, isbreakpnt, accdesc);

    var watchptinfo : WatchpointInfo;
    var ls_match : boolean;
    case DBGWCR(n).LSC[1:0] of
        when '00' => ls_match = FALSE;
        when '01' => ls_match = accdesc.read;
        when '10' => ls_match = accdesc.write || accdesc.acctype == AccessType\_DC;
        when '11' => ls_match = TRUE;
    end;

    var value_match : boolean = FALSE;
    watchptinfo.vaddress = ZeroExtend{64}(vaddress);
    for byte = 0 to size - 1 do
        if (!value_match &&
            !AddressInNaturallyAlignedBlock(watchptinfo.vaddress,
                ZeroExtend{64}(vaddress + byte))) then
            // Watchpoint should report an address which is in
            // the naturally aligned block of the matched address.
            watchptinfo.vaddress = ZeroExtend{64}(vaddress + byte);
        end;
        value_match = value_match || AArch32\_WatchpointByteMatch(n, vaddress + byte);
    end;

    watchptinfo.watchpt_num = n;
    watchptinfo.value_match = value_match && state_match && ls_match && enabled;
    return watchptinfo;
end;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32_Abort

```
// AArch32_Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

func AArch32_Abort(fault : FaultRecord)
begin
    // Check if routed to AArch64 state
    var route_to_aarch64 : boolean = ((IsExternalAbort(fault) &&
        !ELUsingAArch32(SyncExternalAbortTarget(fault))) ||
        (PSTATE.EL == EL0 && !ELUsingAArch32(EL1)));

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2().TGE == '1' || IsSecondStage(fault) ||
            (IsDebugException(fault) && MDCR_EL2().TDE == '1'));
    end;

    if route_to_aarch64 then
        AArch64\_Abort(fault);
    end;
    if fault.accessdesc.acctype == AccessType\_IFETCH then
        AArch32\_TakePrefetchAbortException(fault);
    else
        AArch32\_TakeDataAbortException(fault);
    end;
end;
```


Library pseudocode for aarch32/exceptions/aborts/AArch32_AbortSyndrome

```
// AArch32_AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort exceptions
// taken to Hyp mode
// from an AArch32 translation regime.
func AArch32_AbortSyndrome(exceptype : Exception, fault : FaultRecord,
                           target_el : bits(2)) => ExceptionRecord
begin
    var except : ExceptionRecord = ExceptionSyndrome(exceptype);

    except.syndrome.iss = AArch32\_FaultSyndrome(exceptype, fault);
    if exceptype == Exception\_Watchpoint then
        except.vaddress = fault.watchptinfo.vaddress;
    else
        except.vaddress = ZeroExtend{64}(fault.vaddress);
    end;

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS\_NonSecure then '1' else '0';
        except.ipaddress = ZeroExtend{NUM\_PABITS}(fault.ipaddress.address);
    else
        except.ipavalid = FALSE;
    end;

    return except;
end;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32_CheckPCAlignment

```
// AArch32_CheckPCAlignment()
// =====
func AArch32_CheckPCAlignment()
begin
    let pc : bits(32) = ThisInstrAddr{ }();

    if (CurrentInstrSet() == InstrSet\_A32 && pc[1] == '1') || pc[0] == '1' then
        if AArch32\_GeneralExceptionsToAArch64() then AArch64\_PCAlignmentFault(); end;
        let accdesc : AccessDescriptor = CreateAccDescIfFetch();
        var fault : FaultRecord = NoFault(accdesc, ZeroExtend{64}(pc));
        // Generate an Alignment fault Prefetch Abort exception
        fault.statuscode = Fault\_Alignment;
        AArch32\_Abort(fault);
    end;
end;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32_CommonFaultStatus

```
// AArch32_CommonFaultStatus()
// =====
// Return the common part of the fault status on reporting a Data
// or Prefetch Abort.
func AArch32_CommonFaultStatus(fault : FaultRecord, long_format : boolean) => bits(32)
begin
    var target : bits(32) = Zeros{};
    if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
        let errstate : ErrorState = PEErrorState(fault);
        target[15:14] = AArch32\_EncodeAsyncErrorSyndrome(errstate); // AET
    end;
    if IsExternalAbort(fault) then target[12] = fault.extflag; end; // ExT
    target[9] = if long_format then '1' else '0'; // LPAE
    if long_format then // Long-descriptor format
        target[5:0] = EncodeLDFSC(fault.statuscode, fault.level); // STATUS
    else // Short-descriptor format
        target[10,3:0] = EncodeSDFSC(fault.statuscode, fault.level); // FS
    end;
    return target;
end;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32_ReportDataAbort

```
// AArch32_ReportDataAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

func AArch32_ReportDataAbort(route_to_monitor : boolean, fault : FaultRecord)
begin
    var long_format : boolean;
    if route_to_monitor && CurrentSecurityState() != SS\_Secure then
        long_format = ((TTBCR_S().EAE == '1') ||
            (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR().EAE == '1') ||
                (fault.secondstage && ImpDefBool(
                    "Report abort using Long-descriptor format")))));
    else
        long_format = TTBCR().EAE == '1';
    end;
    var syndrome : bits(32) = AArch32\_CommonFaultStatus(fault, long_format);

    // bits of syndrome that are not common to I and D side
    if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
        syndrome[13] = '1'; // CM
        syndrome[11] = '1'; // WnR
    else
        syndrome[11] = if fault.write then '1' else '0'; // WnR
    end;

    if !long_format then
        syndrome[7:4] = fault.domain; // Domain
    end;

    if fault.accessdesc.acctype == AccessType\_IC then
        var i_syndrome : bits(32);
        if (!long_format &&
            ImpDefBool("Report I-cache maintenance fault in IFSR")) then
            i_syndrome = syndrome;
            syndrome[10,3:0] = EncodeSDFSC(Fault\_ICacheMaint, 1);
        else
            i_syndrome = ARBITRARY : bits(32);
        end;
        if route_to_monitor then
            IFSR_S() = i_syndrome;
        else
            IFSR() = i_syndrome;
        end;
    end;

    if route_to_monitor then
        DFSR_S() = syndrome;
        DFAR_S() = fault.vaddress[31:0];
    else
        DFSR() = syndrome;
        DFAR() = fault.vaddress[31:0];
    end;
end;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32_ReportPrefetchAbort

```
// AArch32_ReportPrefetchAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

func AArch32_ReportPrefetchAbort(route_to_monitor : boolean, fault : FaultRecord)
begin
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    var long_format : boolean = FALSE;
    if route_to_monitor && CurrentSecurityState() != SS\_Secure then
        long_format = TTBCR_S().EAE == '1' || PSTATE.EL == EL2 || TTBCR().EAE == '1';
    else
        long_format = TTBCR().EAE == '1';
    end;

    let fsr : bits(32) = AArch32\_CommonFaultStatus(fault, long_format);
    if route_to_monitor then
        IFSR_S() = fsr;
        IFAR_S() = fault.vaddress[31:0];
    else
        IFSR() = fsr;
        IFAR() = fault.vaddress[31:0];
    end;
    return;
end;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32_TakeDataAbortException

```
// AArch32_TakeDataAbortException()
// =====

func AArch32_TakeDataAbortException(fault : FaultRecord)
begin
    var sea_target : bits(2);
    if IsExternalAbort(fault) then
        sea_target = SyncExternalAbortTarget(fault);
    else
        sea_target = ARBITRARY : bits(2);
    end;

    let route_to_monitor : boolean = IsExternalAbort(fault) && sea_target == EL3;
    let route_to_hyp : boolean      = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                                        (HCR().TGE == '1' ||
                                         (IsExternalAbort(fault) && sea_target == EL2) ||
                                         (IsDebugException(fault) && HCR().TDE == '1') ||
                                         IsSecondStage(fault)));

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x10;
    let lr_offset : integer = 8;

    if IsDebugException(fault) then DBGDSCRext().MOE = fault.debugmoe; end;
    if route_to_monitor then
        AArch32\_ReportDataAbort(route_to_monitor, fault);
        AArch32\_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        let except : ExceptionRecord = AArch32\_AbortSyndrome(Exception\_DataAbort, fault, EL2);
        if PSTATE.EL == EL2 then
            AArch32\_EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32\_EnterHypMode(except, preferred_exception_return, 0x14);
        end;
    else
        AArch32\_ReportDataAbort(route_to_monitor, fault);
        AArch32\_EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32_TakePrefetchAbortException

```
// AArch32_TakePrefetchAbortException()
// =====

func AArch32_TakePrefetchAbortException(fault : FaultRecord)
begin

    var sea_target : bits(2);
    if IsExternalAbort(fault) then
        sea_target = SyncExternalAbortTarget(fault);
    else
        sea_target = ARBITRARY : bits(2);
    end;

    let route_to_monitor : boolean = IsExternalAbort(fault) && sea_target == EL3;
    let route_to_hyp : boolean = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR().TGE == '1' ||
        (IsExternalAbort(fault) && sea_target == EL2) ||
        (IsDebugException(fault) && HDCR().TDE == '1') ||
        IsSecondStage(fault)));

    var except : ExceptionRecord;
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0C;
    let lr_offset : integer = 4;

    if IsDebugException(fault) then DBGDSCRext().MOE = fault.debugmoe; end;
    if route_to_monitor then
        AArch32\_ReportPrefetchAbort(route_to_monitor, fault);
        AArch32\_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault Alignment then // PC Alignment fault
            except = ExceptionSyndrome(Exception PCAlignment);
            except.vaddress = ThisInstrAddr{64}();
        else
            except = AArch32\_AbortSyndrome(Exception InstructionAbort, fault, EL2);
        end;
        if PSTATE.EL == EL2 then
            AArch32\_EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32\_EnterHypMode(except, preferred_exception_return, 0x14);
        end;
    else
        AArch32\_ReportPrefetchAbort(route_to_monitor, fault);
        AArch32\_EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

Library pseudocode for aarch32/exceptions/async/AArch32_TakePhysicalFIQException

```
// AArch32_TakePhysicalFIQException()
// =====

func AArch32_TakePhysicalFIQException()
begin

    // Check if routed to AArch64 state
    var route_to_aarch64 : boolean = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2().TGE == '1' || (HCR_EL2().FMO == '1' && !IsInHost());
    end;

    if !route_to_aarch64 then
        route_to_aarch64 = EffectiveSCR\_EL3\_FIQ() == '1';
    end;

    if route_to_aarch64 then AArch64\_TakePhysicalFIQException(); end;
    let route_to_monitor : boolean = HaveEL(EL3) && SCR().FIQ == '1';
    let route_to_hyp : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR().TGE == '1' || HCR().FMO == '1'));
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x1C;
    let lr_offset : integer = 4;
    if route_to_monitor then
        AArch32\_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        let except : ExceptionRecord = ExceptionSyndrome(Exception FIQ);
        AArch32\_EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32\_EnterMode(M32\_FIQ, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

Library pseudocode for aarch32/exceptions/async/AArch32_TakePhysicalIRQException

```
// AArch32_TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

func AArch32_TakePhysicalIRQException()
begin
    // Check if routed to AArch64 state
    var route_to_aarch64 : boolean = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2().TGE == '1' || (HCR_EL2().IMO == '1' && !IsInHost());
    end;
    if !route_to_aarch64 then
        route_to_aarch64 = EffectiveSCR_EL3_IRQ() == '1';
    end;

    if route_to_aarch64 then AArch64_TakePhysicalIRQException(); end;

    let route_to_monitor : boolean = HaveEL(EL3) && SCR().IRQ == '1';
    let route_to_hyp : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR().TGE == '1' || HCR().IMO == '1'));
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x18;
    let lr_offset : integer = 4;
    if route_to_monitor then
        AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        let except : ExceptionRecord = ExceptionSyndrome(Exception_IRQ);
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32_EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

Library pseudocode for aarch32/exceptions/async/AArch32_TakePhysicalSErrorException

```
// AArch32_TakePhysicalSErrorException()
// =====

func AArch32_TakePhysicalSErrorException(implicit_esb : boolean)
begin
    var masked : boolean;
    var target_el : bits(2);

    (masked, target_el) = PhysicalSErrorTarget();
    assert !masked;

    // Check if routed to AArch64 state
    if !ELUsingAArch32(target_el) then
        AArch64_TakePhysicalSErrorException(implicit_esb);
    end;

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x10;
    let fault : FaultRecord = GetPendingPhysicalSError();
    let lr_offset : integer = 8;

    let except : ExceptionRecord = AArch32_AbortSyndrome(Exception_DataAbort, fault, target_el);
    let route_to_monitor : boolean = (target_el == EL3);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    end;

    case target_el of
        when EL3 =>
            AArch32_ReportDataAbort(route_to_monitor, fault);
            AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        when EL2 =>
            if PSTATE.EL == EL2 then
                AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
            end;
        when EL1 =>
            AArch32_ReportDataAbort(route_to_monitor, fault);
            AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
        otherwise =>
            unreachable;
    end;
end;
```

Library pseudocode for aarch32/exceptions/async/AArch32_TakeVirtualFIQException

```
// AArch32_TakeVirtualFIQException()
// =====

func AArch32_TakeVirtualFIQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR().TGE == '0' && HCR().FMO == '1';
    else
        assert HCR_EL2().TGE == '0' && HCR_EL2().FMO == '1';
    end;
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64_TakeVirtualFIQException(); end;

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x1C;
    let lr_offset : integer = 4;

    AArch32_EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
end;
```

Library pseudocode for aarch32/exceptions/async/AArch32_TakeVirtualIRQException

```
// AArch32_TakeVirtualIRQException()
// =====

func AArch32_TakeVirtualIRQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR().TGE == '0' && HCR().IMO == '1';
    else
        assert HCR_EL2().TGE == '0' && HCR_EL2().IMO == '1';
    end;

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64_TakeVirtualIRQException(); end;

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x18;
    let lr_offset : integer = 4;

    AArch32_EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
end;
```

Library pseudocode for aarch32/exceptions/async/AArch32_TakeVirtualSErrorException

```
// AArch32_TakeVirtualSErrorException()
// =====

func AArch32_TakeVirtualSErrorException()
begin

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR().TGE == '0' && HCR().AMO == '1';
    else
        assert HCR_EL2().TGE == '0' && HCR_EL2().AMO == '1';
    end;
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64_TakeVirtualSErrorException(); end;
    let route_to_monitor : boolean = FALSE;

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x10;
    let lr_offset : integer = 8;

    let vaddress : bits(32) = ARBITRARY : bits(32);
    let parity : boolean = FALSE;
    let fault : Fault = Fault_AsyncExternal;
    let level : integer = ARBITRARY : integer;
    var fsr : bits(32) = Zeros{};
    if IsFeatureImplemented(FEAT_RAS) then
        if ELUsingAArch32(EL2) then
            fsr[15:14] = VDFSR().AET;
            fsr[12] = VDFSR().ExT;
        else
            fsr[15:14] = VSESR_EL2().AET;
            fsr[12] = VSESR_EL2().ExT;
        end;
    else
        fsr[12] = ImpDefBit("Virtual External abort type");
    end;
    if TTBCR().EAE == '1' then // Long-descriptor format
        fsr[9] = '1';
        fsr[5:0] = EncodeLDFSC(fault, level);
    else // Short-descriptor format
        fsr[9] = '0';
        fsr[10,3:0] = EncodeSDFSC(fault, level);
    end;
    DFSR() = fsr;
    DFAR() = ARBITRARY : bits(32);
    ClearPendingVirtualSError();
    AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
end;
```

Library pseudocode for aarch32/exceptions/debug/AArch32_SoftwareBreakpoint

```
// AArch32_SoftwareBreakpoint()
// =====

func AArch32_SoftwareBreakpoint(immediate : bits(16))
begin

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2().TGE == '1' || MDCR_EL2().TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64_SoftwareBreakpoint(immediate);
    end;

    let accdesc : AccessDescriptor = CreateAccDescIFetch();
    var fault : FaultRecord = NoFault(accdesc, ARBITRARY : bits(64));

    fault.statuscode = Fault_Debug;
    fault.debugmoe = DebugException_BKPT;

    AArch32_Abort(fault);
end;
```

Library pseudocode for aarch32/exceptions/debug/DebugException

```
// DebugException
// =====
// Reason codes for debug exceptions, taken to AArch32

constant DebugException_Breakpoint : bits(4) = '0001';
constant DebugException_BKPT : bits(4) = '0011';
constant DebugException_VectorCatch : bits(4) = '0101';
constant DebugException_Watchpoint : bits(4) = '1010';
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32_CheckAdvSIMDOrFPRegisterTraps

```
// AArch32_CheckAdvSIMDOrFPRegisterTraps()
// =====
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

func AArch32_CheckAdvSIMDOrFPRegisterTraps(reg : bits(4))
begin
    if PSTATE.EL == EL1 && EL2Enabled() then
        let tid0 : bit = if ELUsingAArch32(EL2) then HCR().TID0 else HCR_EL2().TID0;
        let tid3 : bit = if ELUsingAArch32(EL2) then HCR().TID3 else HCR_EL2().TID3;
        if ((tid0 == '1' && reg == '0000') || // FPSID
            (tid3 == '1' && reg IN {'0101', '0110', '0111'})) then // MVFRx
            if ELUsingAArch32(EL2) then
                AArch32_SystemAccessTrap(M32_Hyp, 0x8);
            else
                AArch64_AArch32SystemAccessTrap(EL2, 0x8);
            end;
        end;
    end;
end;
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32_ExceptionClass

```
// AArch32_ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in HSR

func AArch32_ExceptionClass(exceptype : Exception) => (integer,bit)
begin
    var il_is_valid : boolean = TRUE;
    var ec : integer;
    case exceptype of
        when Exception Uncategorized => ec = 0x00; il_is_valid = FALSE;
        when Exception WFXTrap => ec = 0x01;
        when Exception CP15RTTrap => ec = 0x03;
        when Exception CP15RRTTrap => ec = 0x04;
        when Exception CP14RTTrap => ec = 0x05;
        when Exception CP14DTTrap => ec = 0x06;
        when Exception AdvSIMDFPAccessTrap => ec = 0x07;
        when Exception FPIDTrap => ec = 0x08;
        when Exception CP14RRTTrap => ec = 0x0C;
        when Exception IllegalState => ec = 0x0E; il_is_valid = FALSE;
        when Exception SupervisorCall => ec = 0x11;
        when Exception HypervisorCall => ec = 0x12;
        when Exception MonitorCall => ec = 0x13;
        when Exception InstructionAbort =>
            ec = if PSTATE.EL == EL2 then 0x21 else 0x20;
            il_is_valid = FALSE;
        when Exception PCAlignment => ec = 0x22; il_is_valid = FALSE;
        when Exception DataAbort => ec = if PSTATE.EL == EL2 then 0x25 else 0x24;
        when Exception FPTrappedException => ec = 0x28;
        when Exception Profiling => ec = 0x3D;
        otherwise => unreachable;
    end;

    var il : bit;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    end;

    return (ec,il);
end;
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32_GeneralExceptionsToAArch64

```
// AArch32_GeneralExceptionsToAArch64()
// =====
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

func AArch32_GeneralExceptionsToAArch64() => boolean
begin
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2().TGE == '1'));
end;
```


Library pseudocode for aarch32/exceptions/exceptions/AArch32_ReportHypEntry

```
// AArch32_ReportHypEntry()
// =====
// Report syndrome information to Hyp mode registers.

func AArch32_ReportHypEntry(except : ExceptionRecord)
begin

    let exceptype : Exception = except.exceptype;

    var (ec,il) : (integer, bit) = AArch32\_ExceptionClass(exceptype);
    let iss : bits(25)= except.syndrome.iss;
    let iss2 : bits(24)= except.syndrome.iss2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss[24] == '0' then
        il = '1';
    end;

    HSR() = ec[5:0]::il::iss;

    if exceptype IN {Exception\_InstructionAbort, Exception\_PCAalignment} then
        HIFAR() = except.vaddress[31:0];
        HDFAR() = ARBITRARY : bits(32);
    elsif exceptype == Exception\_DataAbort then
        HIFAR() = ARBITRARY : bits(32);
        HDFAR() = except.vaddress[31:0];
    end;

    if except.ipavalid then
        HPFAR()[31:4] = except.ipaddress[39:12];
    else
        HPFAR()[31:4] = ARBITRARY : bits(28);
    end;

    return;
end;
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32_ResetControlRegisters

```
// AArch32_ResetControlRegisters()
// =====
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

impdef func AArch32_ResetControlRegisters(cold_reset : boolean)
begin
    return;
end;
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32_TakeReset

```
// AArch32_TakeReset()
// =====
// Reset into AArch32 state

func AArch32_TakeReset(cold_reset : boolean)
begin
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32_WriteMode(M32_Svc);
        SCR().NS = '0'; // Secure state
    elseif HaveEL(EL2) then
        AArch32_WriteMode(M32_Hyp);
    else
        AArch32_WriteMode(M32_Svc);
    end;

    // Reset System registers in the coproc=0b111x encoding space
    // and other system components
    AArch32_ResetControlRegisters(cold_reset);
    FPEXC().EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.[A,I,F] = '111'; // All asynchronous exceptions masked
    PSTATE.IT = '00000000'; // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTLR().TE; // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = HSCTLR().EE; // Endianness: EE=0: little-endian, EE=1: big-endian.
    else
        PSTATE.T = SCTLR().TE; // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = SCTLR().EE; // Endianness: EE=0: little-endian, EE=1: big-endian.
    end;
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32_ResetGeneralRegisters();
    if IsFeatureImplemented(FEAT_SME) || IsFeatureImplemented(FEAT_SVE) then
        ResetSVERegisters();
    else
        AArch32_ResetSIMDFPRegisters();
    end;
    AArch32_ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    var rv : bits(32); // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
        if MVBAR()[0] == '1' then // Reset vector in MVBAR
            rv = MVBAR()[31:1]::'0';
        else
            rv = ImpDefBits{32}("reset vector address");
        end;
    else
        rv = RVBAR()[31:1]::'0';
    end;

    // The reset vector must be correctly aligned
    assert rv[0] == '0' && (PSTATE.T == '1' || rv[1] == '0');

    let branch_conditional : boolean = FALSE;
    EDPRSR().R = '0'; // Leaving Reset State.
    BranchTo{32}(rv, BranchType_RESET, branch_conditional);
end;
```

Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// =====

func ExcVectorBase() => bits(32)
begin
    if SCTLR().V == '1' then // Hivecs selected, base = 0xFFFF0000
        return Ones{16}::Zeros{16};
    else
        return VBAR()[31:5]::Zeros{5};
    end;
end;
```

Library pseudocode for aarch32/exceptions/ieee754/AArch32_FPtrappedException

```
// AArch32_FPtrappedException()
// =====

func AArch32_FPtrappedException(accumulated_exceptions : bits(8))
begin
    if AArch32\_GeneralExceptionsToAArch64() then
        let is_ase : boolean = FALSE;
        let element : integer = 0;
        AArch64\_FPtrappedException(is_ase, accumulated_exceptions);
    end;
    FPEXC().DEX = '1';
    FPEXC().TFV = '1';
    FPEXC()[7,4:0] = accumulated_exceptions[7,4:0]; // IDF,IXF,UFF,OFF,DZF,IOF
    FPEXC()[10:8] = '111'; // VECITR is RES1

    AArch32_TakeUndefInstrException();
end;
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32_CallHypervisor

```
// AArch32_CallHypervisor()
// =====
// Performs a HVC call

func AArch32_CallHypervisor(immediate : bits(16))
begin
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64\_CallHypervisor(immediate);
    else
        AArch32\_TakeHVCEException(immediate);
    end;
end;
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32_CallSupervisor

```
// AArch32_CallSupervisor()
// =====
// Calls the Supervisor

func AArch32_CallSupervisor(immediate_in : bits(16))
begin
    var immediate : bits(16) = immediate_in;
    if CurrentCond() != '1110' then
        immediate = ARBITRARY : bits(16);
    end;
    if AArch32\_GeneralExceptionsToAArch64() then
        AArch64\_CallSupervisor(immediate);
    else
        AArch32\_TakeSVCEException(immediate);
    end;
end;
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32_TakeHVCEException

```
// AArch32_TakeHVCEException()
// =====

func AArch32_TakeHVCEException(immediate : bits(16))
begin
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32\_ITAdvance();
    SSAdvance();
    let preferred_exception_return : bits(32) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x08;

    var except : ExceptionRecord = ExceptionSyndrome(Exception\_HypervisorCall);
    except.syndrome.iss[15:0] = immediate;

    if PSTATE.EL == EL2 then
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
    end;
end;
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32_TakeSMCException

```
// AArch32_TakeSMCException()
// =====

func AArch32_TakeSMCException()
begin
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32_ITAdvance();
    HSAdvance();
    SSAdvance();
    let preferred_exception_return : bits(32) = NextInstrAddr{}();
    let vect_offset : integer = 0x08;
    let lr_offset : integer = 0;
    AArch32_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
end;
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32_TakeSVCEException

```
// AArch32_TakeSVCEException()
// =====

func AArch32_TakeSVCEException(immediate : bits(16))
begin
    AArch32_ITAdvance();
    SSAdvance();
    let route_to_hyp : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR().TGE == '1';

    let preferred_exception_return : bits(32) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x08;
    let lr_offset : integer{} = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        var except : ExceptionRecord = ExceptionSyndrome(Exception_SupervisorCall);
        except.syndrome.iss[15:0] = immediate;
        if PSTATE.EL == EL2 then
            AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
        end;
    else
        AArch32_EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32_EnterHypMode

```
// AArch32_EnterHypMode()
// =====
// Take an exception to Hyp mode.

noreturn func AArch32_EnterHypMode(except : ExceptionRecord, preferred_exception_return : bits(32),
                                   vect_offset : integer)
begin
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS\_NonSecure && ELUsingAArch32(EL2);

    if Halted() then
        AArch32_EnterHypModeInDebugState(except);
    end;
    let spsr : bits(32) = GetPSRFromPSTATE{}(AArch32\_NonDebugState);
    if ! except.excepttype IN {Exception\_IRQ, Exception\_FIQ} then
        AArch32\_ReportHypEntry(except);
    end;
    AArch32\_WriteMode(M32\_Hyp);
    SPSR\_curr() = spsr;
    ELR\_hyp() = preferred_exception_return;
    PSTATE.T = HSCTLR().TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) then
        PSTATE.A = '1';
        PSTATE.I = '1';
        PSTATE.F = '1';
    else
        if ELUsingAArch32(EL3) then
            if SCR().EA == '0' then PSTATE.A = '1'; end;
            if SCR().IRQ == '0' then PSTATE.I = '1'; end;
            if SCR().FIQ == '0' then PSTATE.F = '1'; end;
        else
            if SCR\_EL3().EA == '0' then PSTATE.A = '1'; end;
            if EffectiveSCR\_EL3\_IRQ() == '0' then PSTATE.I = '1'; end;
            if EffectiveSCR\_EL3\_FIQ() == '0' then PSTATE.F = '1'; end;
        end;
    end;
    PSTATE.E = HSCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = HSCTLR().DSSBS; end;
    let branch_conditional : boolean = FALSE;
    BranchTo{32}(HVBAR()[31:5]::vect_offset[4:0], BranchType\_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction(EndOfInstructionReason\_Exception);
end;
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32_EnterMode

```
// AArch32_EnterMode()
// =====
// Take an exception to a mode other than Monitor and Hyp mode.

noreturn func AArch32_EnterMode(target_mode : bits(5),
                                preferred_exception_return : bits(32),
                                lr_offset : integer,
                                vect_offset : integer)
begin
    SynchronizeContext();
    assert ELUsingAArch32\(EL1\) && PSTATE.EL != EL2;

    if Halted\(\) then
        AArch32_EnterModeInDebugState(target_mode);
    end;
    let spsr : bits(32) = GetPSRFromPSTATE{}(AArch32\_NonDebugState);
    if PSTATE.M == M32\_Monitor then SCR().NS = '0'; end;
    AArch32\_WriteMode(target_mode);
    SPSR\_curr() = spsr;
    R\(14\) = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLRL().TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32\_FIQ then
        PSTATE.\[A,I,F\] = '111';
    elsif target_mode IN {M32\_Abort, M32\_IRQ} then
        PSTATE.\[A,I\] = '11';
    else
        PSTATE.I = '1';
    end;
    PSTATE.E = SCTLRL().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) && SCTLRL().SPAN == '0' then PSTATE.PAN = '1'; end;
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLRL().DSSBS; end;
    let branch_conditional : boolean = FALSE;
    BranchTo{32}(ExcVectorBase() [31:5]::vect_offset[4:0], BranchType\_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction(EndOfInstructionReason\_Exception);
end;
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32_EnterMonitorMode

```
// AArch32_EnterMonitorMode()
// =====
// Take an exception to Monitor mode.

func AArch32_EnterMonitorMode(preferred_exception_return : bits(32), lr_offset : integer,
                               vect_offset : integer)
begin
    SynchronizeContext();
    assert HaveEL\(EL3\) && ELUsingAArch32\(EL3\);
    let from_secure : boolean = CurrentSecurityState() == SS\_Secure;
    if Halted\(\) then
        AArch32\_EnterMonitorModeInDebugState();
    end;
    let spsr : bits(32) = GetPSRFromPSTATE{}(AArch32\_NonDebugState);
    if PSTATE.M == M32\_Monitor then SCR().NS = '0'; end;
    AArch32\_WriteMode(M32\_Monitor);
    SPSR\_curr() = spsr;
    R\(14\) = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLRL().TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.\[A,I,F\] = '111';
    PSTATE.E = SCTLRL().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLRL().SPAN == '0' then
            PSTATE.PAN = '1';
        end;
    end;
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLRL().DSSBS; end;
    let branch_conditional : boolean = FALSE;
    BranchTo{32}(MVBAR() [31:5]::vect_offset[4:0], BranchType\_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction(EndOfInstructionReason\_Exception);
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_CheckAdvSIMDOrFPEEnabled

```
// AArch32_CheckAdvSIMDOrFPEEnabled()
// =====
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

func AArch32_CheckAdvSIMDOrFPEEnabled(fpexc_check_in : boolean, advsimd : boolean)
begin
    var fpexc_check : boolean = fpexc_check_in;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        // When executing at EL0 using AArch32, if EL1 is using AArch64 then the Effective value of
        // FPEXC.EN is 1. This includes when EL2 is using AArch64 and enabled in the current
        // Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 1.
        AArch64_CheckFPAdvSIMDEnabled();
    else
        var cpacr_asedis = CPACR().ASEDIS;
        var cpacr_cp10 = CPACR().cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
            // Check if access disabled in NSACR
            if NSACR().NSASEDIS == '1' then cpacr_asedis = '1'; end;
            if NSACR().cp10 == '0' then cpacr_cp10 = '00'; end;
        end;

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then AArch32_Undefined(); end;

            // Check if access disabled in CPACR
            var disabled : boolean;
            case cpacr_cp10 of
                when '00' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0;
                when '10' => disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
                when '11' => disabled = FALSE;
            end;
            if disabled then AArch32_Undefined(); end;
        end;

        // If required, check FPEXC enabled bit.
        if (fpexc_check && PSTATE.EL == EL0 && EL2Enabled() && !ELUsingAArch32(EL2) &&
            HCR_EL2().TGE == '1') then
            // When executing at EL0 using AArch32, if EL2 is using AArch64 and enabled in the
            // current Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 0
            // then it is IMPLEMENTATION DEFINED whether the Effective value of FPEXC.EN is 1
            // or the value of FPEXC32_EL2.EN.
            fpexc_check = ImpDefBool("Use FPEXC32_EL2.EN value when {TGE,RW} == {1,0}");
        end;

        if fpexc_check && FPEXC().EN == '0' then
            AArch32_Undefined();
        end;

        AArch32_CheckFPAdvSIMDTrap(advsimd); // Also check against HCPTR and CPTR_EL3
    end;
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_CheckFPAdvSIMDTrap

```
// AArch32_CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

func AArch32_CheckFPAdvSIMDTrap(advsimd : boolean)
begin
    if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        AArch64\_CheckFPAdvSIMDTrap\(\);
    else
        if (HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) &&
            CPTR_EL3().TFP == '1' && EL3SDDUndefPriority\(\)) then
            Undefined();
        end;

        let ss : SecurityState = CurrentSecurityState\(\);
        if HaveEL\(EL2\) && ss != SS\_Secure then
            var hcptr_tase = HCPTTR().TASE;
            var hcptr_cp10 = HCPTTR().TCP10;

            if HaveEL\(EL3\) && ELUsingAArch32\(EL3\) then
                // Check if access disabled in NSACR
                if NSACR().NSASEDIS == '1' then hcptr_tase = '1'; end;
                if NSACR().cp10 == '0' then hcptr_cp10 = '1'; end;
            end;

            // Check if access disabled in HCPTTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                var except : ExceptionRecord = ExceptionSyndrome(Exception\_AdvSIMDFPAccessTrap);
                except.syndrome.iss[24:20] = ConditionSyndrome\(\);

                if advsimd then
                    except.syndrome.iss[5] = '1';
                else
                    except.syndrome.iss[5] = '0';
                    except.syndrome.iss[3:0] = '1010'; // coproc field, always 0xA
                end;

                if PSTATE.EL == EL2 then
                    AArch32\_TakeUndefInstrException(except);
                else
                    AArch32\_TakeHypTrapException(except);
                end;
            end;
        end;

        if HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3().TFP == '1' then
                if EL3SDDUndef\(\) then
                    Undefined();
                else
                    AArch64\_AdvSIMDFPAccessTrap\(EL3\);
                end;
            end;
        end;
    end;
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_CheckForSMCUndefOrTrap

```
// AArch32_CheckForSMCUndefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

func AArch32_CheckForSMCUndefOrTrap()
begin
    if !HaveEL\(EL3\) || PSTATE.EL == EL0 then
        Undefined();
    end;

    if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        AArch64\_CheckForSMCUndefOrTrap(Zeros{16});
    else
        let route_to_hyp : boolean = EL2Enabled\(\) && PSTATE.EL == EL1 && HCR().TSC == '1';
        if route_to_hyp then
            let except : ExceptionRecord = ExceptionSyndrome(Exception\_MonitorCall);
            AArch32\_TakeHypTrapException(except);
        end;
    end;
end;
```


Library pseudocode for aarch32/exceptions/traps/AArch32_CheckForSVCTrap

```
// AArch32_CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

func AArch32_CheckForSVCTrap(immediate : bits(16))
begin
    if IsFeatureImplemented(FEAT_FGT) then
        let route_to_el2 : boolean = (PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && EL2Enabled() &&
            HFGITR_EL2().SVC_EL0 == '1' &&
            (!IsInHost() && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1')));

        if route_to_el2 then
            var except : ExceptionRecord = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome.iss[15:0] = immediate;
            except.trappedsyscallinst = TRUE;
            let preferred_exception_return : bits(64) = ThisInstrAddr{}();
            let vect_offset : integer = 0x0;

            AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
        end;
    end;
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_CheckForWFXTrap

```
// AArch32_CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

func AArch32_CheckForWFXTrap(target_el : bits(2), wfxtype : WFxType)
begin
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        var trap : boolean;
        let is_wfe : boolean = wfxtype == WFxType_WFE;
        case target_el of
            when EL1 =>
                trap = (if is_wfe then SCTLR_ELx().nTWE else SCTLR_ELx().nTWI) == '0';
            when EL2 =>
                trap = (if is_wfe then HCR_EL2().TWE else HCR_EL2().TWI) == '1';
            when EL3 =>
                trap = (if is_wfe then SCR_EL3().TWE else SCR_EL3().TWI) == '1';
        end;

        if trap then
            if target_el == EL3 && EL3SDDUndef() then
                Undefined();
            else
                AArch64_WFXTrap(wfxtype, target_el);
            end;
        end;
        return;
    end;

    let is_wfe : boolean = wfxtype == WFxType_WFE;
    var trap : boolean;
    case target_el of
        when EL1 =>
            trap = (if is_wfe then SCTLR().nTWE else SCTLR().nTWI) == '0';
        when EL2 =>
            trap = (if is_wfe then HCR().TWE else HCR().TWI) == '1';
        when EL3 =>
            trap = (if is_wfe then SCR().TWE else SCR().TWI) == '1';
    end;

    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2().TGE == '1' then
            AArch64_WFXTrap(wfxtype, target_el);
        end;

        if target_el == EL3 && !EL3SDDUndef() then
            AArch32_TakeMonitorTrapException();
        elseif target_el == EL2 then
            var except : ExceptionRecord = ExceptionSyndrome(Exception_WFXTrap);
            except.syndrome.iss[24:20] = ConditionSyndrome();
            except.syndrome.iss[0] = if wfxtype == WFxType_WFI then '0' else '1';
            AArch32_TakeHypTrapException(except);
        else
            AArch32_TakeUndefInstrException();
        end;
    end;
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_CheckITEnabled

```
// AArch32_CheckITEnabled()
// =====
// Check whether the T32 IT instruction is disabled.

func AArch32_CheckITEnabled(mask : bits(4))
begin
    var it_disabled : bit;

    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR().ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR().ITD else SCTLR_ELx().ITD);
    end;
    if it_disabled == '1' then
        if mask != '1000' then Undefined(); end;

        let accdesc : AccessDescriptor = CreateAccDescIFetch();
        let aligned = TRUE;
        // Otherwise whether the IT block is allowed depends on hwl of the next instruction.
        let next_instr = AArch32_MemSingle{16}(NextInstrAddr{32}(), accdesc, aligned);

        if next_instr IN {'11xxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
                        '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
            Undefined();
        end;
    end;

    return;
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_CheckIllegalState

```
// AArch32_CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

func AArch32_CheckIllegalState()
begin
    if AArch32_GeneralExceptionsToAArch64() then
        AArch64_CheckIllegalState();
    elseif PSTATE.IL == '1' then
        let route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR().TGE == '1';

        let preferred_exception_return : bits(32) = ThisInstrAddr{}();
        let vect_offset : integer = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            let except : ExceptionRecord = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
            end;
        else
            AArch32_TakeUndefInstrException();
        end;
    end;

    return;
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_CheckSETENDEnabled

```
// AArch32_CheckSETENDEnabled()
// =====
// Check whether the AArch32 SETEND instruction is disabled.

func AArch32_CheckSETENDEnabled()
begin
    var setend_disabled : bit;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR().SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR().SED else SCTLR_ELx().SED);
    end;
    if setend_disabled == '1' then
        Undefined();
    end;

    return;
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_SystemAccessTrap

```
// AArch32_SystemAccessTrap()
// =====
// Trapped AArch32 System register access.

func AArch32_SystemAccessTrap(mode : bits(5), ec : integer)
begin
    let (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        let except : ExceptionRecord = AArch32\_SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32\_TakeHypTrapException(except);
    else
        AArch32\_TakeUndefInstrException();
    end;
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_SystemAccessTrapSyndrome

```
// AArch32_SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

func AArch32_SystemAccessTrapSyndrome(instr : bits(32), ec : integer) => ExceptionRecord
begin
    var except : ExceptionRecord;

    case ec of
        when 0x0 => except = ExceptionSyndrome(Exception Uncategorized);
        when 0x3 => except = ExceptionSyndrome(Exception CP15RTTrap);
        when 0x4 => except = ExceptionSyndrome(Exception CP15RRTTrap);
        when 0x5 => except = ExceptionSyndrome(Exception CP14RTTrap);
        when 0x6 => except = ExceptionSyndrome(Exception CP14DTTrap);
        when 0x7 => except = ExceptionSyndrome(Exception AdvSIMDFPAccessTrap);
        when 0x8 => except = ExceptionSyndrome(Exception FPIDTrap);
        when 0xC => except = ExceptionSyndrome(Exception CP14RRTTrap);
        otherwise => unreachable;
    end;

    var iss : bits(20) = Zeros{};

    if except.exceptype == Exception Uncategorized then
        return except;
    elseif except.exceptype IN {Exception FPIDTrap, Exception CP14RTTrap,
                                Exception CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss[13:10] = instr[19:16]; // CRn, Reg in case of VMRS
        iss[8:5] = instr[15:12]; // Rt
        iss[9] = '0'; // RES0

        if except.exceptype != Exception FPIDTrap then // When trap is not for VMRS
            iss[19:17] = instr[7:5]; // opc2
            iss[16:14] = instr[23:21]; // opc1
            iss[4:1] = instr[3:0]; // CRm
        else // VMRS Access
            iss[19:17] = '000'; // opc2 - Hardcoded for VMRS
            iss[16:14] = '111'; // opc1 - Hardcoded for VMRS
            iss[4:1] = '0000'; // CRm - Hardcoded for VMRS
        end;
    elseif except.exceptype IN {Exception CP14RRTTrap, Exception AdvSIMDFPAccessTrap,
                                Exception CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss[19:16] = instr[7:4]; // opc1
        iss[13:10] = instr[19:16]; // Rt2
        iss[8:5] = instr[15:12]; // Rt
        iss[4:1] = instr[3:0]; // CRm
    elseif except.exceptype == Exception CP14DTTrap then
        // Trapped LDC/STC
        iss[19:12] = instr[7:0]; // imm8
        iss[4] = instr[23]; // U
        iss[2:1] = instr[24,21]; // P, W
        if instr[19:16] == '1111' then // Rn==15, LDC(Literal addressing)/STC
            iss[8:5] = ARBITRARY : bits(4);
            iss[3] = '1';
        end;
    end;

    iss[0] = instr[20]; // Direction

    except.syndrome.iss[24:20] = ConditionSyndrome();
    except.syndrome.iss[19:0] = iss;

    return except;
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_TakeHypTrapException

```
// AArch32_TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

func AArch32_TakeHypTrapException(ec : integer)
begin
    let except : ExceptionRecord = AArch32\_SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32\_TakeHypTrapException(except);
end;

// AArch32_TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

func AArch32_TakeHypTrapException(except : ExceptionRecord)
begin
    assert HaveEL(EL2) && CurrentSecurityState() == SS\_NonSecure && ELUsingAArch32(EL2);

    let preferred_exception_return : bits(32) = ThisInstrAddr()();
    let vect_offset : integer = 0x14;

    AArch32\_EnterHypMode(except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_TakeMonitorTrapException

```
// AArch32_TakeMonitorTrapException()
// =====
// Exceptions routed to Monitor mode as a Monitor Trap exception.

func AArch32_TakeMonitorTrapException()
begin
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    let preferred_exception_return : bits(32) = ThisInstrAddr()();
    let vect_offset : integer = 0x04;
    let lr_offset : integer = if CurrentInstrSet() == InstrSet\_A32 then 4 else 2;
    AArch32\_EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_TakeUndefInstrException

```
// AArch32_TakeUndefInstrException()
// =====

noreturn func AArch32_TakeUndefInstrException(except : ExceptionRecord)
begin
    let route_to_hyp : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR().TGE == '1';
    let preferred_exception_return : bits(32) = ThisInstrAddr()();
    let vect_offset : integer = 0x04;
    let lr_offset : integer = if CurrentInstrSet() == InstrSet\_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32\_EnterHypMode(except, preferred_exception_return, vect_offset);
    elseif route_to_hyp then
        AArch32\_EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32\_EnterMode(M32\_Undef, preferred_exception_return, lr_offset, vect_offset);
    end;
end;

// AArch32_TakeUndefInstrException()
// =====

noreturn func AArch32_TakeUndefInstrException()
begin
    let except : ExceptionRecord = ExceptionSyndrome(Exception\_Uncategorized);
    AArch32\_TakeUndefInstrException(except);
end;
```

Library pseudocode for aarch32/exceptions/traps/AArch32_Undefined

```
// AArch32_Undefined()
// =====

noreturn func AArch32_Undefined()
begin
    if AArch32\_GeneralExceptionsToAArch64() then AArch64\_Undefined(); end;
    AArch32\_TakeUndefInstrException();
end;
```

Library pseudocode for aarch32/functions/aborts/AArch32_DomainValid

```
// AArch32_DomainValid()
// =====
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

func AArch32_DomainValid(statuscode : Fault, level : integer) => boolean
begin
    assert statuscode != Fault\_None;

    case statuscode of
        when Fault\_Domain =>
            return TRUE;
        when Fault\_Translation, Fault\_AccessFlag, Fault\_SyncExternalOnWalk, Fault\_SyncParityOnWalk =>
            return level == 2;
        otherwise =>
            return FALSE;
    end;
end;
```

Library pseudocode for aarch32/functions/aborts/AArch32_FaultSyndrome

```
// AArch32_FaultSyndrome()
// =====
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to AArch32 Hyp mode.

func AArch32_FaultSyndrome(exceptype : Exception, fault : FaultRecord) => bits(25)
begin
    assert fault.statuscode != Fault\_None;

    var isstype : IssType;
    isstype.iss = Zeros{25};

    let d_side : boolean = exceptype == Exception\_DataAbort;
    if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
        let errstate : ErrorState = PEErrorState(fault);
        isstype.iss[11:10] = AArch32\_EncodeAsyncErrorSyndrome(errstate); // AET
    end;

    if d_side then

        if AArch32\_InstructionSyndromeValid(fault) then
            isstype.iss[24:14] = LSInstructionSyndrome();
        end;

        if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
            isstype.iss[8] = '1';
        end;

        if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
            isstype.iss[6] = '1';
        elsif fault.statuscode == Fault\_Exclusive then
            isstype.iss[6] = ARBITRARY : bit;
        else
            isstype.iss[6] = if fault.write then '1' else '0';
        end;

    end;

    if IsExternalAbort(fault) then isstype.iss[9] = fault.extflag; end;
    isstype.iss[7] = if fault.s2fslwalk then '1' else '0';
    isstype.iss[5:0] = EncodeLDFSC(fault.statuscode, fault.level);

    return isstype.iss;
end;
```

Library pseudocode for aarch32/functions/aborts/AArch32_InstructionSyndromeValid

```
// AArch32_InstructionSyndromeValid()
// =====
// Returns TRUE if ESR_ELx.ISV is '1' for the given Fault.

func AArch32_InstructionSyndromeValid(fault : FaultRecord) => boolean
begin
    if IsSecondStage(fault) && !fault.s2fslwalk then
        return (!IsExternalSyncAbort(fault) ||
            (!IsFeatureImplemented(FEAT_RAS) && IsExternalAbortOnWalk(fault) &&
                ImpDefBool("ISV on second stage translation table walk")));
    end;

    return FALSE;
end;
```

Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =====
// Function that gives the Short-descriptor FSR code for different types of Fault

func EncodeSDFSC(statuscode : Fault, level : integer) => bits(5)
begin
    var result : bits(5);
    case statuscode of
        when Fault AccessFlag =>
            assert level IN {1,2};
            result = if level == 1 then '00011' else '00110';
        when Fault Alignment =>
            result = '00001';
        when Fault Permission =>
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault Domain =>
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault Translation =>
            assert level IN {1,2};
            result = if level == 1 then '00101' else '00111';
        when Fault SyncExternal =>
            result = '01000';
        when Fault SyncExternalOnWalk =>
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault SyncParity =>
            result = '11001';
        when Fault SyncParityOnWalk =>
            assert level IN {1,2};
            result = if level == 1 then '11100' else '11110';
        when Fault AsyncParity =>
            result = '11000';
        when Fault AsyncExternal =>
            result = '10110';
        when Fault Debug =>
            result = '00010';
        when Fault TLBConflict =>
            result = '10000';
        when Fault Lockdown =>
            result = '10100'; // IMPLEMENTATION DEFINED
        when Fault Exclusive =>
            result = '10101'; // IMPLEMENTATION DEFINED
        when Fault ICacheMaint =>
            result = '00100';
        otherwise =>
            unreachable;
    end;

    return result;
end;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// =====

func A32ExpandImm(imm12 : bits(12)) => bits(32)
begin
    // PSTATE.C argument to following function call does not affect the imm32 result.
    let (imm32, -) = A32ExpandImm\_C(imm12, PSTATE.C);

    return imm32;
end;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm_C

```
// A32ExpandImm_C()
// =====

func A32ExpandImm_C(imm12 : bits(12), carry_in : bit) => (bits(32), bit)
begin
    let unrotated_value : bits(32) = ZeroExtend{}(imm12[7:0]);
    let (imm32, carry_out) : (bits(32), bit) = Shift\_C{32}(unrotated_value, SRTYPE\_ROR,
                                                            2*UInt(imm12[11:8]), carry_in);

    return (imm32, carry_out);
end;
```

Library pseudocode for aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// =====

func DecodeImmShift(srtype : bits(2), imm5 : bits(5)) => (SRTYPE, integer{0..32})
begin
  var shift_t : SRTYPE;
  var shift_n : integer{0..32};
  case srtype of
    when '00' =>
      shift_t = SRTYPE_LSL;  shift_n = UInt(imm5);
    when '01' =>
      shift_t = SRTYPE_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
    when '10' =>
      shift_t = SRTYPE_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
    when '11' =>
      if imm5 == '00000' then
        shift_t = SRTYPE_RRX;  shift_n = 1;
      else
        shift_t = SRTYPE_ROR;  shift_n = UInt(imm5);
      end;
  end;

  return (shift_t, shift_n);
end;
```

Library pseudocode for aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// =====

func DecodeRegShift(srtype : bits(2)) => SRTYPE
begin
  var shift_t : SRTYPE;
  case srtype of
    when '00' => shift_t = SRTYPE_LSL;
    when '01' => shift_t = SRTYPE_LSR;
    when '10' => shift_t = SRTYPE_ASR;
    when '11' => shift_t = SRTYPE_ROR;
  end;
  return shift_t;
end;
```

Library pseudocode for aarch32/functions/common/RRX

```
// RRX()
// =====

func RRX{N}(x : bits(N), carry_in : bit) => bits(N)
begin
  let (result, -) = RRX_C{N}(x, carry_in);
  return result;
end;
```

Library pseudocode for aarch32/functions/common/RRX_C

```
// RRX_C()
// =====

func RRX_C{N}(x : bits(N), carry_in : bit) => (bits(N), bit)
begin
  let result : bits(N) = carry_in :: x[N-1:1];
  let carry_out : bit = x[0];
  return (result, carry_out);
end;
```

Library pseudocode for aarch32/functions/common/SRTYPE

```
// SRTYPE
// =====

type SRTYPE of enumeration {SRTYPE_LSL, SRTYPE_LSR, SRTYPE_ASR, SRTYPE_ROR, SRTYPE_RRX};
```

Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =====

func Shift{N}(value : bits(N), srtype : SRType, amount : integer, carry_in : bit) => bits(N)
begin
    let (result, -) = Shift\_C{N}(value, srtype, amount, carry_in);
    return result;
end;
```

Library pseudocode for aarch32/functions/common/Shift_C

```
// Shift_C()
// =====

func Shift_C{N}(value : bits(N), srtype : SRType, amount : integer,
    carry_in : bit) => (bits(N), bit)
begin
    assert !(srtype == SRType\_RRX && amount != 1);

    var result : bits(N);
    var carry_out : bit;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType\_LSL =>
                (result, carry_out) = LSL_C(value, amount);
            when SRType\_LSR =>
                (result, carry_out) = LSR_C(value, amount);
            when SRType\_ASR =>
                (result, carry_out) = ASR_C(value, amount);
            when SRType\_ROR =>
                (result, carry_out) = ROR_C(value, amount);
            when SRType\_RRX =>
                (result, carry_out) = RRX\_C{N}(value, carry_in);
        end;
    end;

    return (result, carry_out);
end;
```

Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// =====

func T32ExpandImm(imm12 : bits(12)) => bits(32)
begin
    // PSTATE.C argument to following function call does not affect the imm32 result.
    let (imm32, -) = T32ExpandImm\_C(imm12, PSTATE.C);

    return imm32;
end;
```

Library pseudocode for aarch32/functions/common/T32ExpandImm_C

```
// T32ExpandImm_C()
// =====

func T32ExpandImm_C(imm12 : bits(12), carry_in : bit) => (bits(32), bit)
begin
    var imm32 : bits(32);
    var carry_out : bit;
    if imm12[11:10] == '00' then
        case imm12[9:8] of
            when '00' =>
                imm32 = ZeroExtend{32}(imm12[7:0]);
            when '01' =>
                imm32 = '00000000' :: imm12[7:0] :: '00000000' :: imm12[7:0];
            when '10' =>
                imm32 = imm12[7:0] :: '00000000' :: imm12[7:0] :: '00000000';
            when '11' =>
                imm32 = imm12[7:0] :: imm12[7:0] :: imm12[7:0] :: imm12[7:0];
        end;
    end;
    carry_out = carry_in;
else
    let unrotated_value : bits(32) = ZeroExtend{}('1'::imm12[6:0]);
    (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12[11:7]));
end;

return (imm32, carry_out);
end;
```


Library pseudocode for aarch32/functions/common/VBitOps

```
// VBitOps
// =====

type VBitOps of enumeration {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

Library pseudocode for aarch32/functions/common/VCGEType

```
// VCGEType
// =====

type VCGEType of enumeration {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

Library pseudocode for aarch32/functions/common/VCGTtype

```
// VCGTtype
// =====

type VCGTtype of enumeration {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};
```

Library pseudocode for aarch32/functions/common/VFPNegMul

```
// VFPNegMul
// =====

type VFPNegMul of enumeration {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

Library pseudocode for aarch32/functions/coproc/AArch32_CheckCP15InstrCoarseTraps

```
// AArch32_CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

func AArch32_CheckCP15InstrCoarseTraps(CRn : integer, nreg : integer, CRm : integer)
begin
    if PSTATE.EL == EL0 && (!ELUsingAArch32\(EL1\) ||
        (EL2Enabled\(\) && !ELUsingAArch32\(EL2\))) then
        AArch64\_CheckCP15InstrCoarseTraps(CRn, nreg, CRm);
    end;

    let trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
        (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) then
        let major : integer = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR().CRn/HSTR().CRm
        // and MRC and MCR disabled by HCR.TIDCP.
        if (!(major IN {4,14} && HSTR()[major] == '1') ||
            (HCR().TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                ImpDefBool("UNDEF unallocated CP15 access at EL0")) then
                Undefined();
            end;
            if ELUsingAArch32\(EL2\) then
                AArch32\_SystemAccessTrap\(M32\_Hyp, 0x3\);
            else
                AArch64\_AArch32SystemAccessTrap\(EL2, 0x3\);
            end;
        end;
    end;
end;
```

Library pseudocode for aarch32/functions/exclusive/AArch32_ExclusiveMonitorsPass

```
// AArch32_ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give a memory abort.

func AArch32_ExclusiveMonitorsPass(address : bits(32),
                                   size : integer{1, 2, 4, 8, 16, 32}) => boolean
begin
    let acqrel : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                         tagchecked, privileged);

    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    if !AArch32_IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;
    end;

    let memaddrdesc : AddressDescriptor = AArch32_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32_Abort(memaddrdesc.fault);
    end;

    var passed : boolean = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    return passed;
end;
```

Library pseudocode for aarch32/functions/exclusive/AArch32_IsExclusiveVA

```
// AArch32_IsExclusiveVA()
// =====
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

impdef func AArch32_IsExclusiveVA(address : bits(32),
                                   processorid : integer,
                                   size : integer) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for aarch32/functions/exclusive/AArch32_MarkExclusiveVA

```
// AArch32_MarkExclusiveVA()
// =====
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

func AArch32_MarkExclusiveVA(address : bits(32), processorid : integer, size : integer)
begin
    return;
end;
```

Library pseudocode for aarch32/functions/exclusive/AArch32_SetExclusiveMonitors

```
// AArch32_SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

func AArch32_SetExclusiveMonitors(address : bits(32), size : integer{1, 2, 4, 8, 16, 32})
begin
    let acqrel : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp\_LOAD, acqrel,
                                                    tagchecked, privileged);

    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32\_Abort(fault);
    end;

    let memaddrdesc : AddressDescriptor = AArch32\_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;
    end;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32\_MarkExclusiveVA(address, ProcessorID(), size);
end;
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====

func CheckAdvSIMDEnabled()
begin
    let fpexc_check : boolean = TRUE;
    let advsimd : boolean = TRUE;

    AArch32\_CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31 do
        \_Dclone[[i]] = D(i);
    end;

    return;
end;
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEEnabled

```
// CheckAdvSIMDOrVFPEEnabled()
// =====

func CheckAdvSIMDOrVFPEEnabled(include_fpexc_check : boolean, advsimd : boolean)
begin
    AArch32\_CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
end;
```

Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// =====

func CheckCryptoEnabled32()
begin
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
end;
```

Library pseudocode for aarch32/functions/float/CheckVFPEEnabled

```
// CheckVFPEEnabled()
// =====

func CheckVFPEEnabled(include_fpexc_check : boolean)
begin
  let advsimd : boolean = FALSE;
  AArch32\_CheckAdvSIMDOOrFPEEnabled(include_fpexc_check, advsimd);
  // Return from CheckAdvSIMDOOrFPEEnabled() occurs only if VFP access is permitted
  return;
end;
```

Library pseudocode for aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =====

func FPHalvedSub{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  assert N IN {16,32,64};
  let rounding : FPRounding = FPRoundingMode(fpcr);
  let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr);
  let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr);
  var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
  if !done then
    let inf1 : boolean = (type1 == FType\_Infinity);
    let inf2 : boolean = (type2 == FType\_Infinity);
    let zero1 : boolean = (type1 == FType\_Zero);
    let zero2 : boolean = (type2 == FType\_Zero);
    if inf1 && inf2 && sign1 == sign2 then
      result = FPDefaultNaN{N}(fpcr);
      FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
      result = FPInfinity{N}('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
      result = FPInfinity{N}('1');
    elsif zero1 && zero2 && sign1 != sign2 then
      result = FPZero{N}(sign1);
    else
      let result_value : real = (value1 - value2) / 2.0;
      if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        let result_sign : bit = if rounding == FPRounding\_NEGINF then '1' else '0';
        result = FPZero{N}(result_sign);
      else
        result = FPRound{N}(result_value, fpcr);
      end;
    end;
  end;
  return result;
end;
```

Library pseudocode for aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =====

func FPRSqrtStep{N}(op1 : bits(N), op2 : bits(N)) => bits(N)
begin
  assert N IN {16,32};
  let fpcr : FPCR_Type = StandardFPCR();
  let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr);
  let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr);
  var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
  if !done then
    let inf1 : boolean = (type1 == FType\_Infinity);
    let inf2 : boolean = (type2 == FType\_Infinity);
    let zero1 : boolean = (type1 == FType\_Zero);
    let zero2 : boolean = (type2 == FType\_Zero);
    var product : bits(N);
    if (inf1 && zero2) || (zero1 && inf2) then
      product = FPZero{N}('0');
    else
      product = FPMul{N}(op1, op2, fpcr);
    end;
    let three : bits(N) = FPThree{N}('0');
    result = FPHalvedSub{N}(three, product, fpcr);
  end;
  return result;
end;
```

Library pseudocode for aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =====

func FPRecipStep{N}(op1 : bits(N), op2 : bits(N)) => bits(N)
begin
  assert N IN {16,32};
  let fpcr : FPCR_Type = StandardFPCR();
  let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr);
  let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr);
  var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
  if !done then
    let inf1 : boolean = (type1 == FType_Infinity);
    let inf2 : boolean = (type2 == FType_Infinity);
    let zero1 : boolean = (type1 == FType_Zero);
    let zero2 : boolean = (type2 == FType_Zero);
    var product : bits(N);
    if (inf1 && zero2) || (zero1 && inf2) then
      product = FPZero{N}('0');
    else
      product = FPMul{N}(op1, op2, fpcr);
    end;
    let two : bits(N) = FPTwo{N}('0');
    result = FPSub{N}(two, product, fpcr);
  end;
  return result;
end;
```

Library pseudocode for aarch32/functions/float/StandardFPCR

```
// StandardFPCR()
// =====

func StandardFPCR() => FPCR_Type
begin
  let value : bits(32) = ('00000' :: FPSCR().AHP :: '110000' ::
    FPSCR().FZ16 :: '00000000000000000000');
  return ZeroExtend{64}(value);
end;
```

Library pseudocode for aarch32/functions/memory/AArch32_MemSingle

```
// AArch32_MemSingle
// =====

accessor AArch32_MemSingle{size : integer{8, 16, 32, 64}}(address : bits(32),
    accdesc : AccessDescriptor,
    aligned : boolean) <=> value : bits(size)

begin
    // Perform an atomic, little-endian read of 'size' bits.
    getter
        let bytes : integer{} = size DIV 8;
        var value : bits(size);
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        (value, memaddrdesc, memstatus) = AArch32\_MemSingleRead{size}(address, accdesc, aligned);

        // Check for a fault from translation or the output of translation.
        if IsFault(memaddrdesc) then
            AArch32\_Abort(memaddrdesc.fault);
        end;

        // Check for external aborts.
        if IsFault(memstatus) then
            HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, bytes, accdesc);
        end;

        return value;
    end;

    // Perform an atomic, little-endian write of 'size' bits.
    setter
        let bytes : integer{} = size DIV 8;
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        (memaddrdesc, memstatus) = AArch32\_MemSingleWrite{size}(address, accdesc, aligned, value);

        // Check for a fault from translation or the output of translation.
        if IsFault(memaddrdesc) then
            AArch32\_Abort(memaddrdesc.fault);
        end;

        // Check for external aborts.
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;

        return;
    end;
end;
```

Library pseudocode for aarch32/functions/memory/AArch32_MemSingleRead

```
// AArch32_MemSingleRead()
// =====
// Perform an atomic, little-endian read of 'size' bits.

func AArch32_MemSingleRead{size : integer{8, 16, 32, 64}}(address : bits(32),
    accdesc_in : AccessDescriptor,
    aligned : boolean
) => (bits(size), AccessDescriptor,
    PhysMemRetStatus)

begin
    let bytes : integer{} = size DIV 8;
    var value : bits(size) = ARBITRARY : bits(size);
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;
    var accdesc : AccessDescriptor = accdesc_in;
    assert IsAlignedSize(address, bytes);

    var memaddrdesc : AddressDescriptor;
    memaddrdesc = AArch32\_TranslateAddress(address, accdesc, aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (value, memaddrdesc, memstatus);
    end;

    // Memory array access
    (memstatus, value) = PhysMemRead{size}(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        return (value, memaddrdesc, memstatus);
    end;

    if accdesc.acctype == AccessType\_IFETCH then
        if ELUsingAArch32(S1TranslationRegime()) then
            memaddrdesc.fault = AArch32\_CheckDebug(address, accdesc, bytes);
        else
            memaddrdesc.fault = AArch64\_CheckDebug(ZeroExtend{64}(address), accdesc, bytes);
        end;
    end;

    return (value, memaddrdesc, memstatus);
end;
```

Library pseudocode for aarch32/functions/memory/AArch32_MemSingleWrite

```
// AArch32_MemSingleWrite()
// =====
// Perform an atomic, little-endian write of 'size' bits.

func AArch32_MemSingleWrite{size : integer{8, 16, 32, 64}}(address : bits(32),
    accdesc_in : AccessDescriptor,
    aligned : boolean,
    value : bits(size)
) => (AccessDescriptor, PhysMemRetStatus)

begin
    let bytes : integer{} = size DIV 8;
    var accdesc : AccessDescriptor = accdesc_in;
    assert IsAlignedSize(address, bytes);

    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;
    memaddrdesc = AArch32\_TranslateAddress(address, accdesc, aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (memaddrdesc, memstatus);
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
    end;

    memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, value);
    if IsFault(memstatus) then
        return (memaddrdesc, memstatus);
    end;

    return (memaddrdesc, memstatus);
end;
```

Library pseudocode for aarch32/functions/memory/AArch32_UnalignedAccessFaults

```
// AArch32_UnalignedAccessFaults()
// =====
// Determine whether the unaligned access generates an Alignment fault

func AArch32_UnalignedAccessFaults(accdesc : AccessDescriptor) => boolean
begin
    return (AlignmentEnforced() ||
           accdesc.a32lsmd ||
           accdesc.exclusive ||
           accdesc.acqsc ||
           accdesc.relsc);
end;
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadData

```
// Hint_PreloadData()
// =====
// Perform a preload data hint.

impdef func Hint_PreloadData(address : bits(32))
begin
    return;
end;
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadDataForWrite

```
// Hint_PreloadDataForWrite()
// =====
// Perform a preload data hint with a probability that the use will be for a write.

impdef func Hint_PreloadDataForWrite(address : bits(32))
begin
    return;
end;
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadInstr

```
// Hint_PreloadInstr()
// =====
// Perform a preload instruction hint.

impdef func Hint_PreloadInstr(address : bits(32))
begin
    return;
end;
```

Library pseudocode for aarch32/functions/memory/MemA

```
// MemA - accessor
// =====

accessor MemA{size : integer{8, 16, 32, 64}}(address : bits(32)) <=> value : bits(size)
begin
    getter
        let acqrel : boolean = FALSE;
        let privileged : boolean = PSTATE.EL != EL0;
        let tagchecked : boolean = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp\_LOAD, acqrel, tagchecked,
                                                         privileged);

        return Mem\_with\_type(size)(address, accdesc);
    end;

    setter
        let acqrel : boolean = FALSE;
        let privileged : boolean = PSTATE.EL != EL0;
        let tagchecked : boolean = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp\_STORE, acqrel, tagchecked,
                                                         privileged);

        Mem\_with\_type(size)(address, accdesc) = value;
    end;
end;
```


Library pseudocode for aarch32/functions/memory/MemO

```
// MemO
// ====

accessor MemO{size : integer{8, 16, 32, 64} }(address : bits(32)) <=> value : bits(size)
begin
  getter
    let acquire : boolean = TRUE;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp\_LOAD, tagchecked, acquire);
    return Mem with type{size}(address, accdesc);
  end;

  setter
    let acquire : boolean = FALSE;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp\_STORE, tagchecked, acquire);
    Mem with type{size}(address, accdesc) = value;
  end;
end;
```

Library pseudocode for aarch32/functions/memory/MemS

```
// MemS - accessor
// =====

accessor MemS{size : integer{8, 16, 32, 64}}(address : bits(32)) <=> value : bits(size)
begin
  // Memory accessor for streaming load multiple instructions
  getter
    let accdesc : AccessDescriptor = CreateAccDescA32LSMD(MemOp\_LOAD);
    return Mem with type{size}(address, accdesc);
  end;

  // Memory accessor for streaming store multiple instructions
  setter
    let accdesc : AccessDescriptor = CreateAccDescA32LSMD(MemOp\_STORE);
    Mem with type{size}(address, accdesc) = value;
  end;
end;
```

Library pseudocode for aarch32/functions/memory/MemU

```
// MemU - accessor
// =====

accessor MemU{size : integer{8, 16, 32, 64} }(address : bits(32) ) <=> value: bits(size)
begin
  getter
    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp\_LOAD, nontemporal, privileged,
                                                    tagchecked);
    return Mem with type{size}(address, accdesc);
  end;

  setter
    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp\_STORE, nontemporal, privileged,
                                                    tagchecked);
    Mem with type{size}(address, accdesc) = value;
  end;
end;
```

Library pseudocode for aarch32/functions/memory/MemU_unpriv

```
// MemU_unpriv
// =====

accessor MemU_unpriv(size : integer{8, 16, 32, 64})(address : bits(32)) <=> value : bits(size)
begin
  getter
    let nontemporal : boolean = FALSE;
    let privileged : boolean = FALSE;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp\_LOAD, nontemporal, privileged,
                                              tagchecked);

    return Mem\_with\_type(size)(address, accdesc);
  end;

  setter
    let nontemporal : boolean = FALSE;
    let privileged : boolean = FALSE;
    let tagchecked : boolean = FALSE;
    let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp\_STORE, nontemporal, privileged,
                                              tagchecked);

    Mem\_with\_type(size)(address, accdesc) = value;
  end;
end;
```



```

// Mem_with_type
// =====

accessor Mem_with_type(size : integer{8, 16, 32, 64})(address : bits(32),
                                                    accdesc_in : AccessDescriptor
                                                    ) <=> value_in : bits(size)

begin
  // Perform a read of 'size' bits. The access byte order is reversed for a big-endian access.
  // Instruction fetches would call AArch32_MemSingle directly.
  getter
    let bytes : integer{} = size DIV 8;
    var accdesc : AccessDescriptor = accdesc_in;
    var value : bits(size);

    // Check alignment on size of element accessed, not overall access size
    var aligned : boolean;
    if accdesc.ispair then
      let half : integer{} = (size DIV 2) as integer{32};
      aligned = IsAlignedSize(address, half DIV 8);
    else
      aligned = IsAlignedSize(address, bytes);
    end;

    if !aligned && AArch32\_UnalignedAccessFaults(accdesc) then
      let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
      AArch32\_Abort(fault);
    end;
    if aligned then
      value = AArch32\_MemSingle(size)(address, accdesc, aligned);
    else
      assert bytes > 1;
      value[7:0] = AArch32\_MemSingle{8}(address, accdesc, aligned);

      // For subsequent bytes, if they cross to a new translation page which assigns
      // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
      // will generate an Alignment Fault.
      let c : Constraint = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
      assert c IN (Constraint\_FAULT, Constraint\_NONE);
      if c == Constraint\_NONE then aligned = TRUE; end;

      for i = 1 to bytes-1 do
        value[i*8] = AArch32\_MemSingle{8}(address+i, accdesc, aligned);
      end;
    end;

    if BigEndian(accdesc.acctype) then
      value = BigEndianReverse(size)(value);
    end;

    return value;
  end;

  // Perform a write of 'size' bits. The byte order is reversed for a big-endian access.
  setter
    let bytes : integer{} = size DIV 8;
    var value : bits(size) = value_in;
    var accdesc : AccessDescriptor = accdesc_in;

    // Check alignment on size of element accessed, not overall access size
    var aligned : boolean;
    if accdesc.ispair then
      let half : integer{} = (size DIV 2) as integer{32};
      aligned = IsAlignedSize(address, half DIV 8);
    else
      aligned = IsAlignedSize(address, bytes);
    end;

    if !aligned && AArch32\_UnalignedAccessFaults(accdesc) then
      let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
      AArch32\_Abort(fault);
    end;

    if BigEndian(accdesc.acctype) then
      value = BigEndianReverse(size)(value);
    end;
    if aligned then
      AArch32\_MemSingle(size)(address, accdesc, aligned) = value;
    else
      assert bytes > 1;
      AArch32\_MemSingle{8}(address, accdesc, aligned) = value[7:0];

      // For subsequent bytes, if they cross to a new translation page which assigns
      // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
      // will generate an Alignment Fault.

      let c : Constraint = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
      assert c IN (Constraint\_FAULT, Constraint\_NONE);
      if c == Constraint\_NONE then aligned = TRUE; end;
    end;
  end;
end;

```

```

        for i = 1 to bytes-1 do
            AArch32\_MemSingle{8}(address+i, accdesc, aligned) = value[i*:8];
        end;
    end;
    return;
end;
end;

```

Library pseudocode for aarch32/functions/ras/AArch32_ESBOperation

```

// AArch32_ESBOperation()
// =====
// Perform the AArch32 ESB operation for ESB executed in AArch32 state.

func AArch32_ESBOperation()
begin
    var masked : boolean;
    var target_el : bits(2);

    (masked, target_el) = PhysicalErrorTarget();

    // Check if routed to AArch64 state
    if !masked && !ELUsingAArch32(target_el) then
        AArch64\_ESBOperation();
        return;
    end;

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        var syndrome : bits(32) = Zeros{};
        syndrome[31] = '1'; // A
        syndrome[15:0] = AArch32\_PhysicalSErrorSyndrome();
        DISR() = syndrome;
        ClearPendingPhysicalSError();
    end;

    return;
end;

```

Library pseudocode for aarch32/functions/ras/AArch32_EncodeAsyncErrorSyndrome

```

// AArch32_EncodeAsyncErrorSyndrome()
// =====
// Return the encoding for specified ErrorState for an SError exception taken
// to AArch32 state.

func AArch32_EncodeAsyncErrorSyndrome(errorstate : ErrorState) => bits(2)
begin
    case errorstate of
        when ErrorState\_UC => return '00';
        when ErrorState\_UEU => return '01';
        when ErrorState\_UEO => return '10';
        when ErrorState\_UER => return '11';
        otherwise => unreachable;
    end;
end;

```

Library pseudocode for aarch32/functions/ras/AArch32_PhysicalErrorSyndrome

```

// AArch32_PhysicalErrorSyndrome()
// =====
// Generate SError syndrome.

func AArch32_PhysicalErrorSyndrome() => bits(16)
begin
    var syndrome : bits(32) = Zeros{};
    let fault : FaultRecord = GetPendingPhysicalSError();
    if PSTATE.EL == EL2 then
        let errstate : ErrorState = PEErrorState(fault);
        syndrome[11:10] = AArch32\_EncodeAsyncErrorSyndrome(errstate); // AET
        syndrome[9] = fault.extflag; // EA
        syndrome[5:0] = '010001'; // DFSC
    else
        let long_format : boolean = TTBCR().EAE == '1';
        syndrome = AArch32\_CommonFaultStatus(fault, long_format);
    end;
    return syndrome[15:0];
end;

```

Library pseudocode for aarch32/functions/ras/AArch32_vESBOperation

```
// AArch32_vESBOperation()
// =====
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state.
// If FEAT_E3DSE is implemented and there is no unmasked virtual SError exception
// pending, then AArch64_dESBOperation() is called to perform the AArch64 ESB operation
// for a pending delegated SError exception.

func AArch32_vESBOperation()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64\_vESBOperation();
        return;
    end;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a virtual
    // SError interrupt might be pending.
    let vsei_pending : boolean = IsVirtualSErrorPending() && HCR\(\).TGE == '0' && HCR\(\).AMO == '1';
    let vsei_masked : boolean = (PSTATE.A == '1' || Halted() ||
                                ExternalDebugInterruptsDisabled(EL1));

    // Check for a masked virtual SError pending
    if vsei_pending && vsei_masked then
        var syndrome : bits(32) = Zeros{};
        syndrome[31] = '1'; // A
        syndrome[15:14] = VDFSRR\(\).\[15:14\]; // AET
        syndrome[12] = VDFSRR\(\).\[12\]; // ExT
        syndrome[9] = TTBCR\(\).EAE; // LPAE
        if TTBCR\(\).EAE == '1' then // Long-descriptor format
            syndrome[5:0] = '010001'; // STATUS
        else // Short-descriptor format
            syndrome[10,3:0] = '10110'; // FS
        end;
        VDSRR() = syndrome;
        ClearPendingVirtualSError();
    elseif IsFeatureImplemented(FEAT_E3DSE) && !ELUsingAArch32(EL3) then
        AArch64\_dESBOperation();
    end;

    return;
end;
```

Library pseudocode for aarch32/functions/registers/AArch32_ResetGeneralRegisters

```
// AArch32_ResetGeneralRegisters()
// =====

func AArch32_ResetGeneralRegisters()
begin

    for i = 0 to 7 do
        R(i) = ARBITRARY : bits(32);
    end;

    for i = 8 to 12 do
        Rmode(i, M32\_User) = ARBITRARY : bits(32);
        Rmode(i, M32\_FIQ) = ARBITRARY : bits(32);
    end;

    if HaveEL(EL2) then Rmode(13, M32\_Hyp) = ARBITRARY : bits(32); end; // No R14_hyp
    for i = 13 to 14 do
        Rmode(i, M32\_User) = ARBITRARY : bits(32);
        Rmode(i, M32\_FIQ) = ARBITRARY : bits(32);
        Rmode(i, M32\_IRQ) = ARBITRARY : bits(32);
        Rmode(i, M32\_Svc) = ARBITRARY : bits(32);
        Rmode(i, M32\_Abort) = ARBITRARY : bits(32);
        Rmode(i, M32\_Undef) = ARBITRARY : bits(32);
        if HaveEL(EL3) then Rmode(i, M32\_Monitor) = ARBITRARY : bits(32); end;
    end;

    return;
end;
```

Library pseudocode for aarch32/functions/registers/AArch32_ResetSIMDFPRegisters

```
// AArch32_ResetSIMDFPRegisters()
// =====

func AArch32_ResetSIMDFPRegisters()
begin

    for i = 0 to 15 do
        Q(i) = ARBITRARY : bits(128);
    end;

    return;
end;
```

Library pseudocode for aarch32/functions/registers/AArch32_ResetSpecialRegisters

```
// AArch32_ResetSpecialRegisters()
// =====

func AArch32_ResetSpecialRegisters()
begin

    // AArch32 special registers
    SPSR_fiq() [31:0] = ARBITRARY : bits(32);
    SPSR_irq() [31:0] = ARBITRARY : bits(32);
    SPSR_svc() [31:0] = ARBITRARY : bits(32);
    SPSR_abt() [31:0] = ARBITRARY : bits(32);
    SPSR_und() [31:0] = ARBITRARY : bits(32);
    if HaveEL(EL2) then
        SPSR_hyp() = ARBITRARY : bits(32);
        ELR_hyp() = ARBITRARY : bits(32);
    end;
    if HaveEL(EL3) then
        SPSR_mon() = ARBITRARY : bits(32);
    end;

    // External debug special registers
    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);

    return;
end;
```

Library pseudocode for aarch32/functions/registers/AArch32_ResetSystemRegisters

```
// AArch32_ResetSystemRegisters()
// =====

impdef func AArch32_ResetSystemRegisters(cold_reset : boolean)
begin
    return;
end;
```

Library pseudocode for aarch32/functions/registers/ALUEXceptionReturn

```
// ALUEXceptionReturn()
// =====

func ALUEXceptionReturn(address : bits(32))
begin
    if PSTATE.EL == EL2 then
        Undefined();
    elsif PSTATE.M IN {M32_User, M32_System} then
        let c : Constraint = ConstrainUnpredictable(Unpredictable ALUEXCEPTIONRETURN);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP =>
                ExecuteAsNOP();
        end;
    else
        AArch32_ExceptionReturn(address, SPSR_curr());
    end;
end;
```

Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// =====

func ALUWritePC(address : bits(32))
begin
  if CurrentInstrSet\(\) == InstrSet\_A32 then
    BXWritePC(address, BranchType\_INDIR);
  else
    BranchWritePC(address, BranchType\_INDIR);
  end;
end;
```

Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// =====

func BXWritePC(address_in : bits(32), branch_type : BranchType)
begin
  var address : bits(32) = address_in;
  if address[0] == '1' then
    SelectInstrSet(InstrSet\_T32);
    address[0] = '0';
  else
    SelectInstrSet(InstrSet\_A32);
    // For branches to an unaligned PC counter in A32 state, the PE takes the branch
    // and does one of:
    // * Forces the address to be aligned
    // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
    if address[1] == '1' && ConstrainUnpredictableBool(Unpredictable\_A32FORCEALIGNPC) then
      address[1] = '0';
    end;
  end;
  let branch_conditional : boolean = CurrentCond() != '111x';
  BranchTo{32}(address, branch_type, branch_conditional);
end;
```

Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// =====

func BranchWritePC(address_in : bits(32), branch_type : BranchType)
begin
  var address : bits(32) = address_in;
  if CurrentInstrSet() == InstrSet\_A32 then
    address[1:0] = '00';
  else
    address[0] = '0';
  end;
  let branch_conditional : boolean = CurrentCond() != '111x';
  BranchTo{32}(address, branch_type, branch_conditional);
end;
```

Library pseudocode for aarch32/functions/registers/CBWritePC

```
// CBWritePC()
// =====
// Takes a branch from a CBNZ/CBZ instruction.

func CBWritePC(address_in : bits(32))
begin
  var address : bits(32) = address_in;
  assert CurrentInstrSet() == InstrSet\_T32;
  address[0] = '0';
  let branch_conditional : boolean = TRUE;
  BranchTo{32}(address, BranchType\_DIR, branch_conditional);
end;
```


Library pseudocode for aarch32/functions/registers/D

```
// D - accessor
// =====

accessor D(n : integer) <=> value : bits(64)
begin
  getter
    assert n >= 0 && n <= 31;
    let vreg : bits(128) = V{}(n DIVRM 2);
    return vreg[(n MOD 2)*:64];
  end;
  setter
    assert n >= 0 && n <= 31;
    var vreg : bits(128) = V{}(n DIVRM 2);
    vreg[(n MOD 2)*:64] = value;
    V{128}(n DIVRM 2) = vreg;
  end;
end;
```

Library pseudocode for aarch32/functions/registers/Din

```
// Din
// ===
// Return the initial value of D input

func Din(n : integer) => bits(64)
begin
  assert n >= 0 && n <= 31;
  return Dclone[[n]];
end;
```

Library pseudocode for aarch32/functions/registers/H

```
// H - accessor
// =====

accessor H(n : integer) <=> value : bits(16)
begin
  getter
    assert n >= 0 && n <= 31;
    return S(n)[15:0];
  end;
  setter
    S(n) = ZeroExtend{32}(value);
  end;
end;
```

Library pseudocode for aarch32/functions/registers/LR

```
// LR - accessor
// =====

accessor LR() <=> value : bits(32)
begin
  getter
    return R(14);
  end;
  setter
    R(14) = value;
  end;
end;
```

Library pseudocode for aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =====

func LoadWritePC(address : bits(32))
begin
  BXWritePC(address, BranchType\_INDIR);
end;
```

Library pseudocode for aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// =====

func LookUpRIndex(n : integer, mode : bits(5)) => integer
begin
    assert n >= 0 && n <= 14;

    var result : integer;
    case n of // Select index by mode:      usr fiq irq svc abt und hyp
        when 8   => result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
        when 9   => result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
        when 10  => result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11  => result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12  => result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13  => result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14  => result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise => result = n;
    end;

    return result;
end;
```

Library pseudocode for aarch32/functions/registers/Monitor

```
// Monitor mode registers
// =====
// The Monitor mode registers do not map to X registers, so must be defined separately

var SP_mon : bits(32);

var LR_mon : bits(32);
```

Library pseudocode for aarch32/functions/registers/PC32

```
// PC32
// ====
// Read 32-bit program counter

func PC32() => bits(32)
begin
    return R(15); // This includes the offset from AArch32 state
end;
```

Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// =====

func PCStoreValue() => bits(32)
begin
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC32();
end;
```

Library pseudocode for aarch32/functions/registers/Q

```
// Q - accessor
// =====

accessor Q(n : integer) <=> value : bits(128)
begin
    getter
        assert n >= 0 && n <= 15;
        return V{128}(n);
    end;
    setter
        assert n >= 0 && n <= 15;
        V{128}(n) = value;
    end;
end;
```

Library pseudocode for aarch32/functions/registers/Qin

```
// Qin
// ===
// Return the initial value of Q input

func Qin(n : integer) => bits(128)
begin
    assert n >= 0 && n <= 15;
    return Din(2*n+1)::Din(2*n);
end;
```

Library pseudocode for aarch32/functions/registers/R

```
// R - accessor
// =====

accessor R(n : integer) <=> value : bits(32)
begin
    getter
        if n == 15 then
            let offset : integer{} = (if CurrentInstrSet() == InstrSet\_A32 then 8 else 4);
            return PC[31:0] + offset;
        else
            return Rmode(n, PSTATE.M);
        end;
    end;
    setter
        Rmode(n, PSTATE.M) = value;
        return;
    end;
end;

// R - accessor
// =====

accessor R(lr : integer, hr : integer) <=> value : bits(64)
begin
    getter
        return R(hr) :: R(lr);
    end;
    setter
        R(lr) = value[0+:32];
        R(hr) = value[32+:32];
    end;
end;
```

Library pseudocode for aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =====

func RBankSelect(mode : bits(5), usr : integer, fiq : integer, irq : integer,
                 svc : integer, abt : integer, und : integer, hyp : integer) => integer
begin
    var result : integer;
    case mode of
        when M32\_User    => result = usr;    // User mode
        when M32\_FIQ    => result = fiq;    // FIQ mode
        when M32\_IRQ    => result = irq;    // IRQ mode
        when M32\_Svc    => result = svc;    // Supervisor mode
        when M32\_Abort  => result = abt;    // Abort mode
        when M32\_Hyp    => result = hyp;    // Hyp mode
        when M32\_Undef  => result = und;    // Undefined mode
        when M32\_System => result = usr;    // System mode uses User mode registers
        otherwise       => unreachable;    // Monitor mode
    end;

    return result;
end;
```

Library pseudocode for aarch32/functions/registers/ReadAnyAllocatedRegister

```
// ReadAnyAllocatedRegister()
// =====

impdef func ReadAnyAllocatedRegister() => bits(32)
begin
    return Zeros{32};
end;
```

Library pseudocode for aarch32/functions/registers/ReadAnyAllocatedSPSR

```
// ReadAnyAllocatedSPSR()
// =====

impdef func ReadAnyAllocatedSPSR() => bits(32)
begin
    return Zeros{32};
end;
```

Library pseudocode for aarch32/functions/registers/Rmode

```
// Rmode - accessor
// =====

accessor Rmode(n : integer, mode : bits(5)) <=> value : bits(32)
begin
    getter
        assert n >= 0 && n <= 14;

        // Check for attempted use of Monitor mode in Non-secure state.
        if CurrentSecurityState\(\) != SS\_Secure then assert mode != M32\_Monitor; end;
        assert !BadMode(mode);

        if mode == M32\_Monitor then
            if n == 13 then return SP\_mon;
            elsif n == 14 then return LR\_mon;
            else return \_R[[n]][31:0];
            end;
        else
            return \_R[[LookUpRIndex(n, mode)]] [31:0];
        end;
    end;

    setter
        assert n >= 0 && n <= 14;

        // Check for attempted use of Monitor mode in Non-secure state.
        if CurrentSecurityState\(\) != SS\_Secure then assert mode != M32\_Monitor; end;
        assert !BadMode(mode);

        if mode == M32\_Monitor then
            if n == 13 then SP\_mon = value;
            elsif n == 14 then LR\_mon = value;
            else \_R[[n]][31:0] = value;
            end;
        else
            // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
            // register are unchanged or set to zero. This is also tested for on
            // exception entry, as this applies to all AArch32 registers.
            if HaveAArch64\(\) && ConstrainUnpredictableBool(Unpredictable\_ZEROUPPER) then
                \_R[[LookUpRIndex(n, mode)]] = ZeroExtend{64}(value);
            else
                \_R[[LookUpRIndex(n, mode)]] [31:0] = value;
            end;
        end;
    end;
end;
```

Library pseudocode for aarch32/functions/registers/S

```
// S - accessor
// =====

accessor S(n : integer) <=> value : bits(32)
begin
    getter
        assert n >= 0 && n <= 31;
        let vreg : bits(128) = V{}(n DIVRM 4);
        return vreg[(n MOD 4)*:32];
    end;

    setter
        assert n >= 0 && n <= 31;
        var vreg : bits(128) = V{}(n DIVRM 4);
        vreg[(n MOD 4)*:32] = value;
        V{128}(n DIVRM 4) = vreg;
    end;
end;
```

Library pseudocode for aarch32/functions/registers/WriteAnyAllocatedRegister

```
// WriteAnyAllocatedRegister()
// =====

impdef func WriteAnyAllocatedRegister(value : bits(32))
begin
    return;
end;
```

Library pseudocode for aarch32/functions/registers/WriteAnyAllocatedSPSR

```
// WriteAnyAllocatedSPSR()
// =====

impdef func WriteAnyAllocatedSPSR(value : bits(32))
begin
    return;
end;
```

Library pseudocode for aarch32/functions/registers/_Dclone

```
// _Dclone[]
// =====
// Clone the 64-bit Advanced SIMD and VFP extension register bank for use as input to
// instruction pseudocode, to avoid read-after-write for Advanced SIMD and VFP operations.

var _Dclone : array [[32]] of bits(64);
```

Library pseudocode for aarch32/functions/system/AArch32_ExceptionReturn

```
// AArch32_ExceptionReturn()
// =====

func AArch32_ExceptionReturn(new_pc_in : bits(32), spsr : bits(32))
begin
    var new_pc : bits(32) = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR{32}(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc[1:0] = ARBITRARY : bits(2);
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc[0] = '0'; // T32
        else
            new_pc[1:0] = '00'; // A32
        end;
    end;

    let branch_conditional : boolean = CurrentCond() != '111x';
    BranchTo{32}(new_pc, BranchType ERET, branch_conditional);
    CheckExceptionCatch(FALSE); // Check for debug event on exception return
end;
```

Library pseudocode for aarch32/functions/system/AArch32_ExecutingCP10or11Instr

```
// AArch32_ExecutingCP10or11Instr()
// =====

func AArch32_ExecutingCP10or11Instr() => boolean
begin
    let instr : bits(32) = ThisInstr();
    let instr_set : InstrSet = CurrentInstrSet();
    assert instr_set IN {InstrSet A32, InstrSet T32};

    if instr_set == InstrSet A32 then
        return ((instr[27:24] == '1110' || instr[27:25] == '110') && instr[11:8] == '101x');
    else // InstrSet T32
        return (instr[31:28] == '111x' && (instr[27:24] == '1110' || instr[27:25] == '110') &&
            instr[11:8] == '101x');
    end;
end;
```

Library pseudocode for aarch32/functions/system/AArch32_ITAdvance

```
// AArch32_ITAdvance()
// =====

func AArch32_ITAdvance()
begin
    if PSTATE.IT[2:0] == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT[4:0] = LSL(PSTATE.IT[4:0], 1);
    end;
    return;
end;
```

Library pseudocode for aarch32/functions/system/AArch32_InterruptPending

```
// AArch32_InterruptPending()
// =====
// Returns TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise.

func AArch32_InterruptPending() => boolean
begin
    if !ELUsingAArch32(EL2) then
        return AArch64\_InterruptPending();
    end;

    let (irq_pending, -) = IRQPending();
    let (fiq_pending, -) = FIQPending();
    let pending_physical_interrupt : boolean = (irq_pending || fiq_pending ||
        IsPhysicalSErrorPending());

    var pending_virtual_interrupt : boolean = FALSE;
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR\(\).TGE == '0' then
        let virq_pending : boolean = (HCR\(\).IMO == '1' && (VirtualIRQPending() || HCR\(\).VI == '1'));
        let vfiq_pending : boolean = (HCR\(\).FMO == '1' && (VirtualFIQPending() || HCR\(\).VF == '1'));
        let vsei_pending : boolean = (HCR\(\).AMO == '1' &&
            (IsVirtualSErrorPending() || HCR\(\).VA == '1'));

        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;
    end;

    return pending_physical_interrupt || pending_virtual_interrupt;
end;
```

Library pseudocode for aarch32/functions/system/AArch32_SysRegRead

```
// AArch32_SysRegRead()
// =====
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].

impdef func AArch32_SysRegRead(cp_num : integer, instr : bits(32), t : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opcl = instr[21+:3];
    let CRn = instr[16+:4];
    let CRm = instr[0+:4];
    let opc2 = instr[5+:3];
end;
```

Library pseudocode for aarch32/functions/system/AArch32_SysRegRead64

```
// AArch32_SysRegRead64()
// =====
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].

impdef func AArch32_SysRegRead64(cp_num : integer, instr : bits(32), t : integer, t2 : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opcl = instr[4+:4];
    let CRm = instr[0+:4];
end;
```

Library pseudocode for aarch32/functions/system/AArch32_SysRegReadCanWriteAPSR

```
// AArch32_SysRegReadCanWriteAPSR()
// =====
// Determines whether the AArch32 System register read instruction can write to APSR flags.

func AArch32_SysRegReadCanWriteAPSR(cp_num : integer, instr : bits(32)) => boolean
begin
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr[11:8]);

    let opc1 : integer{} = UInt(instr[23:21]);
    let opc2 : integer{} = UInt(instr[7:5]);
    let CRn : integer{} = UInt(instr[19:16]);
    let CRm : integer{} = UInt(instr[3:0]);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;
    end;

    return FALSE;
end;
```

Library pseudocode for aarch32/functions/system/AArch32_SysRegWrite

```
// AArch32_SysRegWrite()
// =====
// Read the contents of R[t] and write to a 32-bit AArch32 System register.

impdef func AArch32_SysRegWrite(cp_num : integer, instr : bits(32), t : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opc1 = instr[21+:3];
    let CRn = instr[16+:4];
    let CRm = instr[ 0+:4];
    let opc2 = instr[ 5+:3];
end;
```

Library pseudocode for aarch32/functions/system/AArch32_SysRegWrite64

```
// AArch32_SysRegWrite64()
// =====
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.

impdef func AArch32_SysRegWrite64(cp_num : integer,
                                   instr : bits(32),
                                   t : integer,
                                   t2 : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opc1 = instr[4+:4];
    let CRm = instr[0+:4];
end;
```

Library pseudocode for aarch32/functions/system/AArch32_SysRegWriteM

```
// AArch32_SysRegWriteM()
// =====
// Read a value from a virtual address and write it to an AArch32 System register.

impdef func AArch32_SysRegWriteM(cp_num : integer, instr : bits(32), address : bits(32))
begin
    let CRd : bits(4) = instr[15:12];
end;
```

Library pseudocode for aarch32/functions/system/AArch32_WriteMode

```
// AArch32_WriteMode()
// =====
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

func AArch32_WriteMode(mode : bits(5))
begin
    let (valid,el) : (boolean, bits(2)) = ELFromM32(mode);
    assert valid;
    PSTATE.M = mode;
    PSTATE.EL = el;
    PSTATE.nRW = '1';
    PSTATE.SP = (if mode IN {M32_User, M32_System} then '0' else '1');
    return;
end;
```

Library pseudocode for aarch32/functions/system/AArch32_WriteModeByInstr

```
// AArch32_WriteModeByInstr()
// =====
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

func AArch32_WriteModeByInstr(mode : bits(5))
begin
    var (valid,el) : (boolean, bits(2)) = ELFromM32(mode);

    // 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;
    end;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32\_Hyp || mode == M32\_Hyp) && PSTATE.M != mode then
        valid = FALSE;
    end;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if (PSTATE.M == M32\_Monitor && HaveEL(EL2) && el == EL1 && SCR().NS == '1'
        && HCR().TGE == '1') then
        valid = FALSE;
    end;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32\_WriteMode(mode);
    end;
end;
```

Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =====

readonly func BadMode(mode : bits(5)) => boolean
begin
    // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
    var valid : boolean;
    case mode of
        when M32\_Monitor =>
            valid = HaveAArch32EL(EL3);
        when M32\_Hyp =>
            valid = HaveAArch32EL(EL2);
        when M32\_FIQ, M32\_IRQ, M32\_Svc, M32\_Abort, M32\_Undef, M32\_System =>
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports EL1 using AArch32
            valid = HaveAArch32EL(EL1);
        when M32\_User =>
            valid = HaveAArch32EL(EL0);
        otherwise =>
            valid = FALSE; // Passed an illegal mode value
    end;
    return !valid;
end;
```


Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

func BankedRegisterAccessValid(SYSm : bits(5), mode : bits(5)) => (boolean, Constraint)
begin
    var valid : boolean = TRUE;
    var c : Constraint = Constraint\_NONE;
    var banked_unpred : boolean = FALSE;
    case SYSm of
        when '000xx', '00100' =>                                // R8_usr to R12_usr
            banked_unpred = mode != M32\_FIQ;
        when '00101' =>                                          // SP_usr
            banked_unpred = mode == M32\_System;
        when '00110' =>                                          // LR_usr
            banked_unpred = mode IN {M32\_Hyp,M32\_System};
        when '010xx', '0110x', '01110' =>                      // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            banked_unpred = mode == M32\_FIQ;
        when '1000x' =>                                          // LR_irq, SP_irq
            banked_unpred = mode == M32\_IRQ;
        when '1001x' =>                                          // LR_svc, SP_svc
            banked_unpred = mode == M32\_Svc;
        when '1010x' =>                                          // LR_abt, SP_abt
            banked_unpred = mode == M32\_Abort;
        when '1011x' =>                                          // LR_und, SP_und
            banked_unpred = mode == M32\_Undef;
        when '1110x' =>                                          // LR_mon, SP_mon
            banked_unpred = (!HaveEL(EL3) || CurrentSecurityState() != SS\_Secure ||
                             mode == M32\_Monitor);
        when '11110' =>                                          // ELR_hyp, only from Monitor or Hyp mode
            banked_unpred = !HaveEL(EL2) || ! mode IN {M32\_Monitor,M32\_Hyp};
        when '11111' =>                                          // SP_hyp, only from Monitor mode
            banked_unpred = !HaveEL(EL2) || mode != M32\_Monitor;
        otherwise =>
            valid = FALSE;
            c = ConstrainUnpredictable(Unpredictable\_UnimplementedRegister);
            assert c IN {Constraint\_UNDEF, Constraint\_NOP, Constraint\_ANYREG};
    end;

    if banked_unpred then
        valid = FALSE;
        c = ConstrainUnpredictable(Unpredictable\_BankedRegister);
        assert c IN {Constraint\_UNDEF, Constraint\_NOP, Constraint\_UNKNOWN};
    end;

    return (valid, c);
end;
```

Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// =====
// Update PSTATE.[N,Z,C,V,Q,GR,E,A,I,F,M] from a CPSR value written by an MSR instruction.

func CPSRWriteByInstr(value : bits(32), bytemask : bits(4))
begin
    let privileged : boolean = PSTATE.EL != ELO; // PSTATE.[A,I,F,M] are not writable at ELO

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask[3] == '1' then
        PSTATE.[N,Z,C,V,Q] = value[31:27];
        // Bits [26:24] are ignored
    end;

    if bytemask[2] == '1' then
        if IsFeatureImplemented(FEAT_SSBS) then
            PSTATE.SSBS = value[23];
        end;
        if privileged then
            PSTATE.PAN = value[22];
        end;
        if IsFeatureImplemented(FEAT_DIT) then
            PSTATE.DIT = value[21];
        end;
        // Bit [20] is RES0
        PSTATE.GE = value[19:16];
    end;

    if bytemask[1] == '1' then
        // Bits [15:10] are RES0
        PSTATE.E = value[9]; // PSTATE.E is writable at ELO
        if privileged then
            PSTATE.A = value[8];
        end;
    end;

    if bytemask[0] == '1' then
        if privileged then
            PSTATE.[I,F] = value[7:6];
            // Bit [5] is RES0
            // AArch32_WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32\_WriteModeByInstr(value[4:0]);
        end;
    end;
    return;
end;
```

Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =====

func ConditionPassed() => boolean
begin
    return ConditionHolds(CurrentCond());
end;
```

Library pseudocode for aarch32/functions/system/CurrentCond

```
// CurrentCond()
// =====
// Returns the condition code that applies to the currently executing instruction

impdef func CurrentCond() => bits(4)
begin
    return Zeros{4};
end;
```

Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()
// =====

func InITBlock() => boolean
begin
    if CurrentInstrSet() == InstrSet\_T32 then
        return PSTATE.IT[3:0] != '0000';
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// =====

func LastInITBlock() => boolean
begin
    return (PSTATE.IT[3:0] == '1000');
end;
```

Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// =====

func SPSRWriteByInstr(value : bits(32), bytemask : bits(4))
begin
    var new_spsr : bits(32) = SPSR_curr();

    if bytemask[3] == '1' then
        new_spsr[31:24] = value[31:24]; // N, Z, C, V, Q flags, IT[1:0], J bits
    end;

    if bytemask[2] == '1' then
        new_spsr[23:16] = value[23:16]; // IL bit, GE[3:0] flags
    end;

    if bytemask[1] == '1' then
        new_spsr[15:8] = value[15:8]; // IT[7:2] bits, E bit, A interrupt mask
    end;

    if bytemask[0] == '1' then
        new_spsr[7:0] = value[7:0]; // I, F interrupt masks, T bit, Mode bits
    end;

    SPSR_curr() = new_spsr; // UNPREDICTABLE if User or System mode

    return;
end;
```

Library pseudocode for aarch32/functions/system/SPSRaccessValid

```
// SPSRaccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

func SPSRaccessValid(SYSm : bits(5), mode : bits(5)) => (boolean, Constraint)
begin
    var valid : boolean = TRUE;
    var c : Constraint = Constraint NONE;
    var banked_unpred : boolean = FALSE;
    case SYSm of
        when '01110' => // SPSR_fiq
            banked_unpred = mode == M32_FIQ;
        when '10000' => // SPSR_irq
            banked_unpred = mode == M32_IRQ;
        when '10010' => // SPSR_svc
            banked_unpred = mode == M32_Svc;
        when '10100' => // SPSR_abt
            banked_unpred = mode == M32_Abort;
        when '10110' => // SPSR_und
            banked_unpred = mode == M32_Undef;
        when '11100' => // SPSR_mon
            banked_unpred = (!HaveEL(EL3) || mode == M32_Monitor ||
                           CurrentSecurityState() != SS_Secure);
        when '11110' => // SPSR_hyp
            banked_unpred = !HaveEL(EL2) || mode != M32_Monitor;
    otherwise =>
        valid = FALSE;
        c = ConstrainUnpredictable(Unpredictable_UnimplementedRegister);
        assert c IN {Constraint UNDEF, Constraint NOP, Constraint ANYREG};
    end;

    if banked_unpred then
        valid = FALSE;
        c = ConstrainUnpredictable(Unpredictable_BankedRegister);
        assert c IN {Constraint UNDEF, Constraint NOP, Constraint UNKNOWN};
    end;

    return (valid, c);
end;
```

Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// =====

func SelectInstrSet(iset : InstrSet)
begin
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_DTLBI_ALL

```
// AArch32_DTLBI_ALL()
// =====
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch32_DTLBI_ALL(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, attr_in : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op = TLBIOp_DALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = UseVMID(regime);
    r.level = TLBILevel_Any;
    r.attr = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_DTLBI_ASID

```
// AArch32_DTLBI_ASID()
// =====
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_DTLBI_ASID(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, attr_in : TLBIMemAttr, Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op = TLBIOp_DASID;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = UseVMID(regime);
    r.level = TLBILevel_Any;
    r.attr = attr;
    r.asid = Zeros{8} :: Rt[7:0];

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_DTLBI_VA

```
// AArch32_DTLBI_VA()
// =====
// Invalidate by VA all stage 1 data TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_DTLBI_VA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32\_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op = TLBIOp\_DVA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = UseVMID(regime);
    r.level = level;
    r.attr = attr;
    r.asid = Zeros{8} :: Rt[7:0];
    r.address = Zeros{32} :: Rt[31:12] :: Zeros{12};

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_EffectiveBroadcast

```
// AArch32_EffectiveBroadcast()
// =====

func AArch32_EffectiveBroadcast(broadcast_in : Broadcast) => Broadcast
begin
    var broadcast : Broadcast = broadcast_in;

    if PSTATE.EL != EL1 then return broadcast; end;

    if (broadcast == Broadcast\_NSH && ((ELUsingAArch32(EL2) && HCR().FB == '1') ||
        (!ELUsingAArch32(EL2) && HCR_EL2().FB == '1')) then
        broadcast = Broadcast\_ForcedISH;
    end;

    return broadcast;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_ITLBI_ALL

```
// AArch32_ITLBI_ALL()
// =====
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch32_ITLBI_ALL(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                      broadcast_in : Broadcast, attr_in : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32\_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op = TLBIOp\_IALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = UseVMID(regime);
    r.level = TLBILevel\_Any;
    r.attr = attr;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_ITLBI_ASID

```
// AArch32_ITLBI_ASID()
// =====
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID
// (where regime supports) and ASID in the parameter Rt in the indicated translation
// regime with the indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_ITLBI_ASID(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                      broadcast_in : Broadcast, attr_in : TLBIMemAttr, Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32\_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op = TLBIOp\_IASID;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = UseVMID(regime);
    r.level = TLBILevel\_Any;
    r.attr = attr;
    r.asid = Zeros{8} :: Rt[7:0];

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_ITLBI_VA

```
// AArch32_ITLBI_VA()
// =====
// Invalidate by VA all stage 1 instruction TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_ITLBI_VA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32\_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op = TLBIOp\_IVA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = UseVMID(regime);
    r.level = level;
    r.attr = attr;
    r.asid = Zeros{8} :: Rt[7:0];
    r.address = Zeros{32} :: Rt[31:12] :: Zeros{12};

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_TLBI_ALL

```
// AArch32_TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch32_TLBI_ALL(security : SecurityState, regime : Regime,
    broadcast : Broadcast, attr : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2};

    var r : TLBIRecord;
    r.op = TLBIOp\_ALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.attr = attr;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_TLBI_ASID

```
// AArch32_TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_TLBI_ASID(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                      broadcast_in : Broadcast, attr_in : TLBIMemAttr, Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32\_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op          = TLBIOp\_ASID;
    r.from_aarch64 = FALSE;
    r.security    = security;
    r.regime      = regime;
    r.vmid        = vmid;
    r.use_vmid    = UseVMID(regime);
    r.level       = TLBILevel\_Any;
    r.attr        = attr;
    r.asid        = Zeros{8} :: Rt[7:0];

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_TLBI_IPAS2

```
// AArch32_TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.

func AArch32_TLBI_IPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                      broadcast : Broadcast, level : TLBILevel, attr : TLBIMemAttr, Rt : bits(32))
begin
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS\_NonSecure;

    var r : TLBIRecord;
    r.op          = TLBIOp\_IPAS2;
    r.from_aarch64 = FALSE;
    r.security    = security;
    r.regime      = regime;
    r.vmid        = vmid;
    r.use_vmid    = TRUE;
    r.level       = level;
    r.attr        = attr;
    r.address     = Zeros{24} :: Rt[27:0] :: Zeros{12};
    r.ipaspace    = PAS\_NonSecure;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```


Library pseudocode for aarch32/functions/tlbi/AArch32_TLBI_VA

```
// AArch32_TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_TLBI_VA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Rt : bits(32))

begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32\_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op = TLBIOp\_VA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = UseVMID(regime);
    r.level = level;
    r.attr = attr;
    r.asid = Zeros{8} :: Rt[7:0];
    r.address = Zeros{32} :: Rt[31:12] :: Zeros{12};

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_TLBI_VAA

```
// AArch32_TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_TLBI_VAA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Rt : bits(32))

begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32\_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op = TLBIOp\_VAA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = UseVMID(regime);
    r.level = level;
    r.attr = attr;
    r.address = Zeros{32} :: Rt[31:12] :: Zeros{12};

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_TLBI_VMALL

```
// AArch32_TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

func AArch32_TLBI_VMALL(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, attr_in : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var attr : TLBIMemAttr = attr_in;

    let broadcast : Broadcast = AArch32\_EffectiveBroadcast(broadcast_in);
    if ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var r : TLBIRecord;
    r.op = TLBIOp\_VMALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.vmid = vmid;
    r.use_vmid = UseVMID(regime);
    r.attr = attr;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/tlbi/AArch32_TLBI_VMALLS12

```
// AArch32_TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// broadcast domain that match the indicated VMID.

func AArch32_TLBI_VMALLS12(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast : Broadcast, attr : TLBIMemAttr)
begin
    assert PSTATE.EL IN {EL3, EL2};

    var r : TLBIRecord;
    r.op = TLBIOp\_VMALLS12;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.vmid = vmid;
    r.use_vmid = TRUE;
    r.attr = attr;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

func Sat{N}(i : integer, unsigned : boolean) => bits(N)
begin
    let result : bits(N) = if unsigned then UnsignedSat{N}(i) else SignedSat{N}(i);
    return result;
end;
```

Library pseudocode for aarch32/ic/AArch32_CanTrapIC

```
// AArch32_CanTrapIC()
// =====
// Determines whether the execution of the IC instruction can be trapped.

func AArch32_CanTrapIC(cacheop : CacheOp, opscope : CacheOpScope) => boolean
begin
    return (!AArch32\_TreatICAsNOP(cacheop, opscope) || ImpDefBool(
        "When IC is treated as NOP, data cache maintenance operations are trapped"));
end;
```

Library pseudocode for aarch32/ic/AArch32_IC

```
// AArch32_IC()
// =====
// Perform Instruction Cache Operation.

func AArch32_IC(opscope : CacheOpScope)
begin
    let regval : bits(32) = ARBITRARY : bits(32);
    AArch32\_IC(regval, opscope);
end;

// AArch32_IC()
// =====
// Perform Instruction Cache Operation.

func AArch32_IC(regval : bits(32), opscope : CacheOpScope)
begin
    var cache : CacheRecord;

    cache.acctype = AccessType\_IC;
    cache.cachetype = CacheType\_Instruction;
    cache.cacheop = CacheOp\_Invalidate;
    cache.opscope = opscope;
    cache.security = SecurityStateAtEL(PSTATE.EL);

    if opscope IN {CacheOpScope\_ALLU, CacheOpScope\_ALLUIS} then
        if (opscope == CacheOpScope\_ALLUIS ||
            AArch32\_EffectiveBroadcast(Broadcast\_NSH) == Broadcast\_ForcedISH) then
            cache.shareability = Shareability\_ISH;
        else
            cache.shareability = Shareability\_NSH;
        end;
        cache.regval = ZeroExtend{64}(regval);
        CACHE\_OP(cache);
    else
        assert opscope == CacheOpScope\_PoU;

        if EL2Enabled() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid = VMID();
            else
                cache.is_vmid_valid = FALSE;
            end;
        else
            cache.is_vmid_valid = FALSE;
        end;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid = ASID();
        else
            cache.is_asid_valid = FALSE;
        end;

        cache.shareability = Shareability\_NSH;
        cache.vaddress = ZeroExtend{64}(regval);

        let size : integer = 0;
        let aligned : boolean = TRUE;
        let accdesc : AccessDescriptor = CreateAccDescIC(cache);
        let memaddrdesc : AddressDescriptor = AArch32\_TranslateAddress(regval, accdesc, aligned,
                                                                    size);

        if IsFault(memaddrdesc) then
            AArch32\_Abort(memaddrdesc.fault);
        end;

        cache.paddress = memaddrdesc.paddress;
        CACHE\_OP(cache);
    end;
    return;
end;
```

Library pseudocode for aarch32/ic/AArch32_TreatICAsNOP

```
// AArch32_TreatICAsNOP()
// =====
// Determines whether the execution of the IC instruction is treated as a NOP.

func AArch32_TreatICAsNOP(cacheop : CacheOp, opscope : CacheOpScope) => boolean
begin
    if CTR().DIC == '1' then
        return ImpDefBool("IC is treated as NOP");
    end;

    return FALSE;
end;
```

Library pseudocode for aarch32/predictionrestrict/AArch32_RestrictPrediction

```
// AArch32_RestrictPrediction()
// =====
// Clear all predictions in the context.

func AArch32_RestrictPrediction(val : bits(32), restriction : RestrictType)
begin

    var c : ExecutionCntxt;
    let target_el : bits(2) = val[25:24];

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then ExecuteAsNOP(); end;

    let ns : bit = val[26];
    let nse : bit = ARBITRARY : bit;
    let ss : SecurityState = TargetSecurityState(ns, nse);

    c.security = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if PSTATE.EL IN {ELO, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid = FALSE;
            c.vmid = VMID();

            elsif target_el IN {ELO, EL1} then
                c.is_vmid_valid = TRUE;
                c.all_vmid = val[27] == '1';
                c.vmid = ZeroExtend{NUM_VMIDBITS}(val[23:16]); // Only valid if val[27] == '0'
            else
                c.is_vmid_valid = FALSE;
            end;
        else
            c.is_vmid_valid = FALSE;
        end;

        if PSTATE.EL == ELO then
            c.is_asid_valid = TRUE;
            c.all_asid = FALSE;
            c.asid = ASID();

            elsif target_el == ELO then
                c.is_asid_valid = TRUE;
                c.all_asid = val[8] == '1';
                c.asid = ZeroExtend{NUM_ASIDBITS}(val[7:0]); // Only valid if val[8] == '0'
            else
                c.is_asid_valid = FALSE;
            end;

            c.restriction = restriction;
            RESTRICT_PREDICTIONS(c);
        end;
```



```

// AArch32_DefaultTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, without TEX remap

func AArch32_DefaultTEXDecode(TEX_in : bits(3), c_in : bit, b_in : bit, s : bit) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;
    var TEX : bits(3) = TEX_in;
    var c : bit = c_in;
    var b : bit = b_in;

    // Reserved values map to allocated values
    if (TEX == '001' && c::b == '01') || (TEX == '010' && c::b != '00') || TEX == '011' then
        var texcb : bits(5);
        (-, texcb) = ConstrainUnpredictableBits{5}(Unpredictable RESTEXCB);
        TEX = texcb[4:2]; c = texcb[1]; b = texcb[0];
    end;

    // Distinction between Inner Shareable and Outer Shareable is not supported in this format
    // A memory region is either Non-shareable or Outer Shareable
    case TEX::c::b of
        when '00000' =>
            // Device-nGnRnE
            memattrs.memtype = MemType Device;
            memattrs.device = DeviceType nGnRnE;
            memattrs.shareability = Shareability OSH;
        when '00001', '01000' =>
            // Device-nGnRE
            memattrs.memtype = MemType Device;
            memattrs.device = DeviceType nGnRE;
            memattrs.shareability = Shareability OSH;
        when '00010' =>
            // Write-through Read allocate
            memattrs.memtype = MemType Normal;
            memattrs.inner.attrs = MemAttr WT;
            memattrs.inner.hints = MemHint RA;
            memattrs.outer.attrs = MemAttr WT;
            memattrs.outer.hints = MemHint RA;
            memattrs.shareability = if s == '1' then Shareability OSH else Shareability NSH;
        when '00011' =>
            // Write-back Read allocate
            memattrs.memtype = MemType Normal;
            memattrs.inner.attrs = MemAttr WB;
            memattrs.inner.hints = MemHint RA;
            memattrs.outer.attrs = MemAttr WB;
            memattrs.outer.hints = MemHint RA;
            memattrs.shareability = if s == '1' then Shareability OSH else Shareability NSH;
        when '00100' =>
            // Non-cacheable
            memattrs.memtype = MemType Normal;
            memattrs.inner.attrs = MemAttr NC;
            memattrs.outer.attrs = MemAttr NC;
            memattrs.shareability = Shareability OSH;
        when '00110' =>
            memattrs = ImpDefMemoryAttributes("IMPLEMENTATION_DEFINED");
        when '00111' =>
            // Write-back Read and Write allocate
            memattrs.memtype = MemType Normal;
            memattrs.inner.attrs = MemAttr WB;
            memattrs.inner.hints = MemHint RWA;
            memattrs.outer.attrs = MemAttr WB;
            memattrs.outer.hints = MemHint RWA;
            memattrs.shareability = if s == '1' then Shareability OSH else Shareability NSH;
        when '1xxxx' =>
            // Cacheable, TEX[1:0] = Outer attrs, {c,b} = Inner attrs
            memattrs.memtype = MemType Normal;
            memattrs.inner = DecodeSDFAttr(c::b);
            memattrs.outer = DecodeSDFAttr(TEX[1:0]);

            if memattrs.inner.attrs == MemAttr NC && memattrs.outer.attrs == MemAttr NC then
                memattrs.shareability = Shareability OSH;
            else
                memattrs.shareability = if s == '1' then Shareability OSH else Shareability NSH;
            end;
        otherwise =>
            // Reserved, handled above
            unreachable;
    end;

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tags = MemTag Untagged;

    if memattrs.inner.attrs == MemAttr WB && memattrs.outer.attrs == MemAttr WB then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';
    end;
end;

```

```
    return memattrs;  
end;
```

Library pseudocode for aarch32/translation/attrs/AArch32_MAIRAttr

```
// AArch32_MAIRAttr()  
// =====  
// Retrieve the memory attribute encoding indexed in the given MAIR  
  
func AArch32_MAIRAttr(index : integer, mair : bits(64)) => bits(8)  
begin  
    assert (index < 8);  
    return mair[index*8];  
end;
```

Library pseudocode for aarch32/translation/atrrs/AArch32_RemappedTEXDecode

```
// AArch32_RemappedTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, with TEX remap

func AArch32_RemappedTEXDecode(regime : Regime, TEX : bits(3),
                               c : bit, b : bit, s : bit) => MemoryAttributes
begin
    var mematrrs : MemoryAttributes;
    var prrr : PRRR\_Type;
    var nmrr : NMRR\_Type;

    let region : integer{} = UInt(TEX[0]::c::b); // TEX[2:1] are ignored in this mapping scheme
    if region == 6 then
        return ImpDefMemoryAttributes("IMPLEMENTATION_DEFINED");
    end;

    if regime == Regime\_EL30 then
        prrr = PRRR\_S\(\);
        nmrr = NMRR\_S\(\);
    elsif HaveEL\(EL3\) && ELUsingAArch32\(EL3\) then
        prrr = PRRR\_NS\(\);
        nmrr = NMRR\_NS\(\);
    else
        prrr = PRRR\(\);
        nmrr = NMRR\(\);
    end;

    let base : integer{} = 2 * region;
    var attrfield : bits(2) = prrr[region*2];

    if attrfield == '11' then // Reserved, maps to allocated value
        (-, attrfield) = ConstrainUnpredictableBits{2}(Unpredictable\_RESPRRR);
    end;

    case attrfield of
        when '00' => // Device-nGnRnE
            mematrrs.memtype = MemType\_Device;
            mematrrs.device = DeviceType\_nGnRnE;
            mematrrs.shareability = Shareability\_OSH;
        when '01' => // Device-nGnRE
            mematrrs.memtype = MemType\_Device;
            mematrrs.device = DeviceType\_nGnRE;
            mematrrs.shareability = Shareability\_OSH;
        when '10' =>
            let NSn : bit = if s == '0' then prrr.NS0 else prrr.NS1;
            let NOSm : bit = prrr[region+24] AND NSn;
            let IRn : bits(2) = nmrr[base+1:base];
            let ORn : bits(2) = nmrr[base+17:base+16];

            mematrrs.memtype = MemType\_Normal;
            mematrrs.inner = DecodeSDFAttr(IRn);
            mematrrs.outer = DecodeSDFAttr(ORn);
            if mematrrs.inner.attrs == MemAttr\_NC && mematrrs.outer.attrs == MemAttr\_NC then
                mematrrs.shareability = Shareability\_OSH;
            else
                let sh : bits(2) = NSn::NOSm;
                mematrrs.shareability = DecodeShareability(sh);
            end;
        when '11' =>
            unreachable;
    end;

    // The Transient hint is not supported in this format
    mematrrs.inner.transient = FALSE;
    mematrrs.outer.transient = FALSE;
    mematrrs.tags = MemTag\_Untagged;

    if mematrrs.inner.attrs == MemAttr\_WB && mematrrs.outer.attrs == MemAttr\_WB then
        mematrrs.xs = '0';
    else
        mematrrs.xs = '1';
    end;

    return mematrrs;
end;
```


Library pseudocode for aarch32/translation/debug/AArch32_CheckBreakpoint

```
// AArch32_CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

func AArch32_CheckBreakpoint(fault_in : FaultRecord, vaddress : bits(32),
                             accdesc : AccessDescriptor, size : integer) => FaultRecord
begin
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert size IN {2,4};

    var fault : FaultRecord = fault_in;
    var match : boolean = FALSE;
    var mismatch : boolean = FALSE;
    var brkptinfo : BreakpointInfo;

    for i = 0 to NumBreakpointsImplemented\(\) - 1 do
        brkptinfo = AArch32\_BreakpointMatch(i, vaddress, accdesc, size);
        match = match || brkptinfo.match;
        mismatch = mismatch || brkptinfo.mismatch;
    end;

    if match && HaltOnBreakpointOrWatchpoint\(\) then
        let reason : bits(6) = DebugHalt\_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) then
        fault.statuscode = Fault\_Debug;
        fault.debugmoe = DebugException\_Breakpoint;
        fault.vaddress = ZeroExtend{64}(vaddress);
    end;

    return fault;
end;
```

Library pseudocode for aarch32/translation/debug/AArch32_CheckDebug

```
// AArch32_CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

func AArch32_CheckDebug(vaddress : bits(32), accdesc : AccessDescriptor,
                         size : integer) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, ZeroExtend{64}(vaddress));

    let d_side : boolean = IsWatchpointableAccess(accdesc);
    let i_side : boolean = (accdesc.acctype == AccessType\_IFETCH);
    let generate_exception : boolean = (AArch32\_GenerateDebugExceptions\(\) &&
                                         DBGDSCRext\(\).MDBGen == '1');

    let halt : boolean = HaltOnBreakpointOrWatchpoint\(\);
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    let vector_catch_first : boolean = ConstrainUnpredictableBool(Unpredictable\_BPVECTORCATCHPRI);

    if i_side && vector_catch_first && generate_exception then
        fault = AArch32\_CheckVectorCatch(fault, vaddress, size);
    end;

    if fault.statuscode == Fault\_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32\_CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch32\_CheckBreakpoint(fault, vaddress, accdesc, size);
        end;
    end;

    if fault.statuscode == Fault\_None && i_side && !vector_catch_first && generate_exception then
        return AArch32\_CheckVectorCatch(fault, vaddress, size);
    end;

    return fault;
end;
```

Library pseudocode for aarch32/translation/debug/AArch32_CheckVectorCatch

```
// AArch32_CheckVectorCatch()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

func AArch32_CheckVectorCatch(fault_in : FaultRecord, vaddress : bits(32),
                             size : integer) => FaultRecord
begin
    assert ELUsingAArch32(S1TranslationRegime());

    var fault : FaultRecord = fault_in;
    var match : boolean = AArch32\_VCRMATCH(vaddress);
    if size == 4 && !match && AArch32\_VCRMATCH(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHHALF);
    end;

    if match then
        fault.statuscode = Fault\_Debug;
        fault.debugmoe = DebugException\_VectorCatch;
    end;

    return fault;
end;
```

Library pseudocode for aarch32/translation/debug/AArch32_CheckWatchpoint

```
// AArch32_CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

func AArch32_CheckWatchpoint(fault_in : FaultRecord, vaddress : bits(32),
                             accdesc : AccessDescriptor, size : integer) => FaultRecord
begin
    assert ELUsingAArch32(S1TranslationRegime());
    var fault : FaultRecord = fault_in;
    var match : boolean = FALSE;

    if !IsWatchpointableAccess(accdesc) then
        return fault;
    end;

    for i = 0 to NumWatchpointsImplemented() - 1 do
        let watchptinfo : WatchpointInfo = AArch32\_WatchpointMatch(i, vaddress, size, accdesc);
        match = match || watchptinfo.value_match;
    end;

    if match && HaltOnBreakpointOrWatchpoint() then
        let reason : bits(6) = DebugHalt\_Watchpoint;
        EDWAR() = ZeroExtend{64}(vaddress);
        Halt(reason);
    elseif match then
        fault.statuscode = Fault\_Debug;
        fault.debugmoe = DebugException\_Watchpoint;
        fault.vaddress = ZeroExtend{64}(vaddress);
    end;

    return fault;
end;
```

Library pseudocode for aarch32/translation/faults/AArch32_IPAIsOutOfRange

```
// AArch32_IPAIsOutOfRange()
// =====
// Check intermediate physical address bits not resolved by translation are ZERO

func AArch32_IPAIsOutOfRange(walkparams : S2TTWParams, ipa : bits(40)) => boolean
begin
    // Input Address size
    let iasize : integer{} = AArch32\_S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa[39:iasize]);
end;
```

Library pseudocode for aarch32/translation/faults/AArch32_S1HasAlignmentFaultDueToMemType

```
// AArch32_S1HasAlignmentFaultDueToMemType()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses to memory type

func AArch32_S1HasAlignmentFaultDueToMemType(accdesc : AccessDescriptor, aligned : boolean,
                                             ntlsmd : bit, memattrs : MemoryAttributes) => boolean
begin
    if memattrs.memtype != MemType\_Device then
        return FALSE;
    elsif accdesc.a32lsmd && ntlsmd == '0' then
        return memattrs.device != DeviceType\_GRE;
    elsif accdesc.acctype == AccessType\_DCZero then
        return TRUE;
    elsif !aligned then
        return !(ImpDefBool("Device location supports unaligned access"));
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch32/translation/faults/AArch32_S1LDHasPermissionsFault

```
// AArch32_S1LDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

func AArch32_S1LDHasPermissionsFault(regime : Regime, walkparams : S1TTWParams, perms : Permissions,
                                     memtype : MemType, paspace : PASpace,
                                     accdesc : AccessDescriptor) => boolean

begin
    var r, w, x : bit;
    var pr, pw : bit;
    var ur, uw : bit;
    var xn : bit;
    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case perms.ap[2:1] of
            when '00' => (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' => (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' => (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' => (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
        end;

        // Apply hierarchical permissions
        case perms.ap_table of
            when '00' => (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' => (pr,pw,ur,uw) = ( pr, pw, '0','0'); // Privileged access
            when '10' => (pr,pw,ur,uw) = ( pr, '0', ur, '0'); // Read-only
            when '11' => (pr,pw,ur,uw) = ( pr, '0', '0', '0'); // Read-only, privileged access
        end;

        xn = perms.xn OR perms.xn_table;
        let pxn : bit = perms.pxn OR perms.pxn_table;

        let ux : bit = ur AND NOT(xn OR (uw AND walkparams.wxn));
        let px : bit = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));

        if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
            let pan : bit = PSTATE.PAN AND (ur OR uw);
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);
        end;

        (r,w,x) = if accdesc.el == ELO then (ur,uw,ux) else (pr,pw,px);

        // Prevent execution from Non-secure space by PE in Secure state if SIF is set
        if accdesc.ss == SS\_Secure && paspace == PAS\_NonSecure then
            x = x AND NOT(walkparams.sif);
        end;
    else
        // Apply leaf permissions
        case perms.ap[2] of
            when '0' => (r,w) = ('1','1'); // No effect
            when '1' => (r,w) = ('1','0'); // Read-only
        end;

        // Apply hierarchical permissions
        case perms.ap_table[1] of
            when '0' => (r,w) = ( r, w ); // No effect
            when '1' => (r,w) = ( r, '0'); // Read-only
        end;

        xn = perms.xn OR perms.xn_table;
        x = NOT(xn OR (w AND walkparams.wxn));
    end;

    if accdesc.acctype == AccessType\_IFETCH then
        let constraint : Constraint = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
        if constraint == Constraint\_FAULT && memtype == MemType\_Device then
            return TRUE;
        else
            return x == '0';
        end;
    elsif accdesc.acctype IN {AccessType\_IC, AccessType\_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
    end;
end;
```

Library pseudocode for aarch32/translation/faults/AArch32_S1SDHasPermissionsFault

```
// AArch32_S1SDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

func AArch32_S1SDHasPermissionsFault(regime : Regime, perms_in : Permissions, memtype : MemType,
                                     paspace : PASpace, accdesc : AccessDescriptor) => boolean
begin
    var perms : Permissions = perms_in;
    var pr, pw : bit;
    var ur, uw : bit;
    var sctlr : SCTLr_Type;
    if regime == Regime\_EL30 then
        sctlr = SCTLr_S();
    elseif HaveEL(EL3) && ELUsingAArch32(EL3) then
        sctlr = SCTLr_NS();
    else
        sctlr = SCTLr();
    end;

    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = ImpDefBits{3}("Reserved short descriptor AP encoding");
        end;

        case perms.ap of
            when '000' => (pr,pw,ur,uw) = ('0','0','0','0'); // No access
            when '001' => (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '010' => (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, RO at PL0
            when '011' => (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            // '100' is reserved
            when '101' => (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '110' => (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL (deprecated)
            when '111' => (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
        end;
    else // Simplified access permissions model
        case perms.ap[2:1] of
            when '00' => (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' => (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' => (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' => (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
        end;
    end;

    let ux : bit = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    let px : bit = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw AND sctlr.UWXN));

    if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
        let pan : bit = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);
    end;

    var (r,w,x) : (bit, bit, bit) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if accdesc.ss == SS\_Secure && paspace == PAS\_NonSecure then
        x = x AND NOT(if ELUsingAArch32(EL3) then SCR().SIF else SCR_EL3().SIF);
    end;

    if accdesc.acctype == AccessType\_IFETCH then
        if (memtype == MemType\_Device &&
            ConstrainUnpredictable(Unpredictable\_INSTRDEVICE) == Constraint\_FAULT) then
            return TRUE;
        else
            return x == '0';
        end;
    elseif accdesc.acctype IN {AccessType\_IC, AccessType\_DC} then
        return FALSE;
    elseif accdesc.write then
        return w == '0';
    else
        return r == '0';
    end;
end;
```

Library pseudocode for aarch32/translation/faults/AArch32_S2HasAlignmentFaultDueToMemType

```
// AArch32_S2HasAlignmentFaultDueToMemType()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses due to memory type

func AArch32_S2HasAlignmentFaultDueToMemType(accdesc : AccessDescriptor, aligned : boolean,
                                             memattrs : MemoryAttributes) => boolean
begin
    if memattrs.memtype != MemType\_Device then
        return FALSE;
    elsif accdesc.acctype == AccessType\_DCZero then
        return TRUE;
    elsif !aligned then
        return !(ImpDefBool("Device location supports unaligned access"));
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch32/translation/faults/AArch32_S2HasPermissionsFault

```
// AArch32_S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

func AArch32_S2HasPermissionsFault(walkparams : S2TTWParams, perms : Permissions, memtype : MemType,
                                   accdesc : AccessDescriptor) => boolean
begin
    let r : bit = perms.s2ap[0];
    let w : bit = perms.s2ap[1];
    var x : bit;
    var ux : bit;
    var px : bit;
    if IsFeatureImplemented(FEAT_XNX) then
        case perms.s2xn::perms.s2xnx of
            when '00' => (px, ux) = ( r, r );
            when '01' => (px, ux) = ('0', r );
            when '10' => (px, ux) = ('0', '0');
            when '11' => (px, ux) = ( r, '0');
        end;

        x = if accdesc.el == ELO then ux else px;
    else
        x = r AND NOT(perms.s2xn);
    end;

    if accdesc.acctype == AccessType\_TTW then
        return (walkparams.ptw == '1' && memtype == MemType\_Device) || r == '0';
    elsif accdesc.acctype == AccessType\_IFETCH then
        let constraint : Constraint = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
        return (constraint == Constraint\_FAULT && memtype == MemType\_Device) || x == '0';
    elsif accdesc.acctype IN {AccessType\_IC, AccessType\_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
    end;
end;
```

Library pseudocode for aarch32/translation/faults/AArch32_S2InconsistentSL

```
// AArch32_S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

func AArch32_S2InconsistentSL(walkparams : S2TTWParams) => boolean
begin
    let startlevel : integer = AArch32\_S2StartLevel(walkparams.sl0);
    let levels : integer = FINAL\_LEVEL - startlevel;
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let stride : integer{} = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    let sl_min_iasize : integer = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits // Bits directly mapped to output address
        + 1); // At least 1 more bit to be decoded by initial level

    // Can accommodate 1 more stride in the level + concatenation of up to 2^4 tables
    let sl_max_iasize : integer = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    let iasize : AddressSize = AArch32\_S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
end;
```

Library pseudocode for aarch32/translation/faults/AArch32_VAIsOutOfRange

```
// AArch32_VAIsOutOfRange()
// =====
// Check virtual address bits not resolved by translation are identical
// and of accepted value

func AArch32_VAIsOutOfRange(regime : Regime, walkparams : S1TTWParams, va : bits(32)) => boolean
begin
    if regime == Regime\_EL2 then
        // Input Address size
        let iasize : integer{} = AArch32\_S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va[31:iasize]);
    elseif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        let lo_iasize : integer{} = AArch32\_S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        let up_iasize : integer{} = AArch32\_S1IASize(walkparams.t1sz);
        return !IsZero(va[31:lo_iasize]) && !IsOnes(va[31:up_iasize]);
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32_GetS1TLBContext

```
// AArch32_GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLB entries

func AArch32_GetS1TLBContext(regime : Regime, ss : SecurityState, va : bits(32)) => TLBContext
begin
    var tlbcontext : TLBContext;

    case regime of
        when Regime\_EL2 => tlbcontext = AArch32\_TLBContextEL2(va);
        when Regime\_EL10 => tlbcontext = AArch32\_TLBContextEL10(ss, va);
        when Regime\_EL30 => tlbcontext = AArch32\_TLBContextEL30(va);
    end;

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;
end;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32_GetS2TLBContext

```
// AArch32_GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB entries

func AArch32_GetS2TLBContext(ipa : FullAddress) => TLBContext
begin
    assert ipa.paspace == PAS_NonSecure;

    var tlbcontext : TLBContext;

    tlbcontext.ss      = SS_NonSecure;
    tlbcontext.regime  = Regime_EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid    = ZeroExtend{NUM_VMIDBITS}(VTTBR().VMID);
    tlbcontext.tg      = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia       = ZeroExtend{64}(ipa.address);
    tlbcontext.cnp      = if IsFeatureImplemented(FEAT_TTCNP) then VTTBR().CnP else '0';

    return tlbcontext;
end;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32_TLBContextEL10

```
// AArch32_TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime
// (PL10 when EL3 is A64) to match against TLB entries

func AArch32_TLBContextEL10(ss : SecurityState, va : bits(32)) => TLBContext
begin
    var tlbcontext : TLBContext;
    var ttbcr : TTBCR_Type;
    var ttbr0 : TTBR0_Type;
    var ttbr1 : TTBR1_Type;
    var contextidr : CONTEXTIDR_Type;

    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        ttbcr = TTBCR_NS();
        ttbr0 = TTBR0_NS();
        ttbr1 = TTBR1_NS();
        contextidr = CONTEXTIDR_NS();
    else
        ttbcr = TTBCR();
        ttbr0 = TTBR0();
        ttbr1 = TTBR1();
        contextidr = CONTEXTIDR();
    end;

    tlbcontext.ss = ss;
    tlbcontext.regime = Regime_EL10;

    if AArch32_EL2Enabled(ss) then
        tlbcontext.vmid = ZeroExtend{NUM_VMIDBITS}(VTTBR().VMID);
    end;

    if ttbcr.EAE == '1' then
        if ttbcr.A1 == '0' then
            tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(ttbr0.ASID);
        else
            tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(ttbr1.ASID);
        end;
    else
        tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(contextidr.ASID);
    end;

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend{64}(va);

    if IsFeatureImplemented(FEAT_TTCNP) && ttbcr.EAE == '1' then
        if AArch32_GetVARange(va, ttbcr.TOSZ, ttbcr.TISZ) == VARange_LOWER then
            tlbcontext.cnp = ttbr0.CnP;
        else
            tlbcontext.cnp = ttbr1.CnP;
        end;
    else
        tlbcontext.cnp = '0';
    end;

    return tlbcontext;
end;
```


Library pseudocode for aarch32/translation/tlbcontext/AArch32_TLBContextEL2

```
// AArch32_TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries

func AArch32_TLBContextEL2(va : bits(32)) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss      = SS\_NonSecure;
    tlbcontext.regime = Regime\_EL2;
    tlbcontext.ia      = ZeroExtend{64}(va);
    tlbcontext.tg       = TGx\_4KB;
    tlbcontext.cnp      = if IsFeatureImplemented(FEAT_TTCNP) then HTTBR().CnP else '0';

    return tlbcontext;
end;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32_TLBContextEL30

```
// AArch32_TLBContextEL30()
// =====
// Gather translation context for accesses under EL30 regime
// (PL10 in Secure state and EL3 is A32) to match against TLB entries

func AArch32_TLBContextEL30(va : bits(32)) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss      = SS\_Secure;
    tlbcontext.regime = Regime\_EL30;
    if TTBCR_S().EAE == '1' then
        tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(if TTBCR_S().A1 == '0' then TTBR0_S().ASID
                                                    else TTBR1_S().ASID);
    else
        tlbcontext.asid = ZeroExtend{NUM_ASIDBITS}(CONTEXTIDR_S().ASID);
    end;

    tlbcontext.tg = TGx\_4KB;
    tlbcontext.ia = ZeroExtend{64}(va);

    if IsFeatureImplemented(FEAT_TTCNP) && TTBCR_S().EAE == '1' then
        if AArch32\_GetVARange(va, TTBCR_S().TOSZ, TTBCR_S().T1SZ) == VARange\_LOWER then
            tlbcontext.cnp = TTBR0_S().CnP;
        else
            tlbcontext.cnp = TTBR1_S().CnP;
        end;
    else
        tlbcontext.cnp = '0';
    end;

    return tlbcontext;
end;
```

Library pseudocode for aarch32/translation/translation/AArch32_EL2Enabled

```
// AArch32_EL2Enabled()
// =====
// Returns whether EL2 is enabled for the given Security State

func AArch32_EL2Enabled(ss : SecurityState) => boolean
begin
    if ss == SS\_Secure then
        if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2)) then
            return FALSE;
        elsif HaveEL(EL3) then
            return SCR_EL3().EEL2 == '1';
        else
            return ImpDefBool("Secure-only implementation");
        end;
    else
        return HaveEL(EL2);
    end;
end;
```

Library pseudocode for aarch32/translation/translation/AArch32_FullTranslate

```
// AArch32_FullTranslate()
// =====
// Perform address translation as specified by VMSA-A32

func AArch32_FullTranslate(va : bits(32), accdesc : AccessDescriptor,
                          aligned : boolean) => AddressDescriptor
begin
    // Prepare fault fields in case a fault is detected
    var fault : FaultRecord = NoFault(accdesc, ZeroExtend{64}(va));
    let regime : Regime = TranslationRegime(accdesc.el);

    // First Stage Translation
    var ipa : AddressDescriptor;
    if regime == Regime\_EL2 || TTBCR().EAE == '1' then
        (fault, ipa) = AArch32\_S1TranslateLD(fault, regime, va, aligned, accdesc);
    else
        (fault, ipa, -) = AArch32\_S1TranslateSD(fault, regime, va, aligned, accdesc);
    end;

    if fault.statuscode != Fault\_None then
        return CreateFaultyAddressDescriptor(fault);
    end;

    if regime == Regime\_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend{64}(va);
        var pa : AddressDescriptor;
        (fault, pa) = AArch32\_S2Translate(fault, ipa, aligned, accdesc);

        if fault.statuscode != Fault\_None then
            return CreateFaultyAddressDescriptor(fault);
        else
            return pa;
        end;
    else
        return ipa;
    end;
end;
```

Library pseudocode for aarch32/translation/translation/AArch32_OutputDomain

```
// AArch32_OutputDomain()
// =====
// Determine the domain the translated output address

func AArch32_OutputDomain(regime : Regime, domain : bits(4)) => bits(2)
begin
    var Dn : bits(2);
    if regime == Regime\_EL30 then
        Dn = DACR_S() [UInt(domain)*:2];
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        Dn = DACR_NS() [UInt(domain)*:2];
    else
        Dn = DACR() [UInt(domain)*:2];
    end;

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits{2}(Unpredictable\_RESDACR);
    end;

    return Dn;
end;
```



```

// AArch32_S1DisabledOutput()
// =====
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

func AArch32_S1DisabledOutput(fault_in : FaultRecord, regime : Regime,
                             va : bits(32), aligned : boolean,
                             accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
begin

    var fault : FaultRecord = fault_in;
    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    var memattrs : MemoryAttributes;
    var default_cacheable : bit;
    if regime == Regime\_EL10 && AArch32\_EL2Enabled(accdesc.ss) then
        if ELStateUsingAArch32(EL2, accdesc.ss == SS\_Secure) then
            default_cacheable = HCR().DC;
        else
            default_cacheable = HCR_EL2().DC;
        end;
    else
        default_cacheable = '0';
    end;

    if default_cacheable == '1' then
        // Use default cacheable settings
        memattrs.memtype = MemType\_Normal;
        memattrs.inner.attrs = MemAttr\_WB;
        memattrs.inner.hints = MemHint\_RWA;
        memattrs.outer.attrs = MemAttr\_WB;
        memattrs.outer.hints = MemHint\_RWA;
        memattrs.shareability = Shareability\_NSH;
        if (EL2Enabled() && !ELStateUsingAArch32(EL2, accdesc.ss == SS\_Secure) &&
            IsFeatureImplemented(FEAT_MTE2) && HCR_EL2().DCT == '1') then
            memattrs.tags = MemTag\_AllocationTagged;
        else
            memattrs.tags = MemTag\_Untagged;
        end;
        memattrs.xs = '0';
    elsif accdesc.acctype == AccessType\_IFETCH then
        memattrs.memtype = MemType\_Normal;
        memattrs.shareability = Shareability\_OSH;
        memattrs.tags = MemTag\_Untagged;
        if AArch32\_S1ICacheEnabled(regime) then
            memattrs.inner.attrs = MemAttr\_WT;
            memattrs.inner.hints = MemHint\_RA;
            memattrs.outer.attrs = MemAttr\_WT;
            memattrs.outer.hints = MemHint\_RA;
        else
            memattrs.inner.attrs = MemAttr\_NC;
            memattrs.outer.attrs = MemAttr\_NC;
        end;
        memattrs.xs = '1';
    else
        // Treat memory region as Device
        memattrs.memtype = MemType\_Device;
        memattrs.device = DeviceType\_nGnRnE;
        memattrs.shareability = Shareability\_OSH;
        memattrs.tags = MemTag\_Untagged;
        memattrs.xs = '1';
    end;

    var ntlsmd : bit;
    if IsFeatureImplemented(FEAT_LSMAOC) then
        case regime of
            when Regime\_EL30 => ntlsmd = SCTLR_S().nTLSMD;
            when Regime\_EL2 => ntlsmd = HSCTLR().nTLSMD;
            when Regime\_EL10 =>
                if HaveEL(EL3) && ELUsingAArch32(EL3) then
                    ntlsmd = SCTLR_NS().nTLSMD;
                else
                    ntlsmd = SCTLR().nTLSMD;
                end;
        end;
    else
        ntlsmd = '1';
    end;

    if AArch32\_S1HasAlignmentFaultDueToMemType(accdesc, aligned, ntlsmd, memattrs) then
        fault.statuscode = Fault\_Alignment;
        return (fault, ARBITRARY : AddressDescriptor);
    elsif (accdesc.exclusive &&
        (! (regime == Regime\_EL10 && EL2Enabled() &&
            (!ELUsingAArch32(EL2) && HCR_EL2().DC == '1') ||
            (ELUsingAArch32(EL2) && HCR().DC == '1')))) &&
        ConstrainUnpredictableBool(Unpredictable\_Atomic\_MMU\_IMPDEF\_FAULT)) then
        fault.statuscode = Fault\_Exclusive;
    end;
end;

```

```

var oa : FullAddress;
oa.address = ZeroExtend{NUM\_PABITS}(va);
oa.paspace = if accdesc.ss == SS\_Secure then PAS\_Secure else PAS\_NonSecure;
let ipa : AddressDescriptor = CreateAddressDescriptor(ZeroExtend{64}(va),
                                                    oa,
                                                    memattrs,
                                                    accdesc);

return (fault, ipa);
end;

```

Library pseudocode for aarch32/translation/translation/AArch32_S1Enabled

```

// AArch32_S1Enabled()
// =====
// Returns whether stage 1 translation is enabled for the active translation regime

func AArch32_S1Enabled(regime : Regime, ss : SecurityState) => boolean
begin
    if regime == Regime\_EL2 then
        return HSCTLR().M == '1';
    elsif regime == Regime\_EL30 then
        return SCTLR_S().M == '1';
    elsif !AArch32\_EL2Enabled(ss) then
        return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTLR_NS().M else SCTLR().M) == '1';
    elsif ELStateUsingAArch32(EL2, ss == SS\_Secure) then
        if HaveEL(EL3) && ELUsingAArch32(EL3) then
            return HCR().[TGE,DC] == '00' && SCTLR_NS().M == '1';
        else
            return HCR().[TGE,DC] == '00' && SCTLR().M == '1';
        end;
    else
        return EL2Enabled() && HCR_EL2().[TGE,DC] == '00' && SCTLR().M == '1';
    end;
end;
end;

```

Library pseudocode for aarch32/translation/translation/AArch32_S1TranslateLD

```
// AArch32_S1TranslateLD()
// =====
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime

func AArch32_S1TranslateLD(fault_in : FaultRecord, regime : Regime,
                           va : Bits(32), aligned : boolean,
                           accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
begin
    var fault : FaultRecord = fault_in;

    if !AArch32\_S1Enabled(regime, accdesc.ss) then
        return AArch32\_S1DisabledOutput(fault, regime, va, aligned, accdesc);
    end;

    let walkparams : S1TTWParams = AArch32\_GetS1TTWParams(regime, va);

    if AArch32\_VAIsOutOfRange(regime, walkparams, va) then
        fault.level = 1;
        fault.statuscode = Fault\_Translation;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var walkstate : TTWState;
    (fault, walkstate) = AArch32\_S1WalkLD(fault, regime, walkparams, accdesc, va);

    if fault.statuscode != Fault\_None then
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

    if AArch32\_S1HasAlignmentFaultDueToMemType(accdesc, aligned, walkparams.ntlsmid,
                                                walkstate.memattrs) then
        fault.statuscode = Fault\_Alignment;
    elseif AArch32\_S1LDHasPermissionsFault(regime, walkparams,
                                             walkstate.permissions,
                                             walkstate.memattrs.memtype,
                                             walkstate.baseaddress.paspace,
                                             accdesc) then
        fault.statuscode = Fault\_Permission;
    end;

    if fault.statuscode != Fault\_None then
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var memattrs : MemoryAttributes;
    if ((accdesc.acctype == AccessType\_IFETCH &&
        (walkstate.memattrs.memtype == MemType\_Device || !AArch32\_S1ICacheEnabled(regime))) ||
        (accdesc.acctype != AccessType\_IFETCH &&
        walkstate.memattrs.memtype == MemType\_Normal && !AArch32\_S1DCacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    else
        memattrs = walkstate.memattrs;
    end;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime\_EL10 && AArch32\_EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS\_Secure) then HCR().VM
         else HCR_EL2().VM) == '1' &&
         !(ImpDefBool("Apply effective shareability at stage 1"))) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);
    end;

    // Output Address
    let oa : FullAddress = StageOA(ZeroExtend{64}(va), walkparams.d128, walkparams.tgx, walkstate);
    let ipa : AddressDescriptor = CreateAddressDescriptor(ZeroExtend{64}(va),
                                                         oa, memattrs, accdesc);

    return (fault, ipa);
end;
```



```

// AArch32_S1TranslateSD()
// =====
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
// depending on the regime

func AArch32_S1TranslateSD(fault in : FaultRecord, regime : Regime,
                           va : bits(32), aligned : boolean,
                           accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor, SDFType)

begin
    var fault : FaultRecord = fault_in;

    if !AArch32\_S1Enabled(regime, accdesc.ss) then
        var ipa : AddressDescriptor;
        (fault, ipa) = AArch32\_S1DisabledOutput(fault, regime, va, aligned, accdesc);
        return (fault, ipa, ARBITRARY : SDFType);
    end;

    var walkstate : TTWState;
    (fault, walkstate) = AArch32\_S1WalkSD(fault, regime, accdesc, va);

    if fault.statuscode != Fault\_None then
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : SDFType);
    end;

    let domain : bits(2) = AArch32\_OutputDomain(regime, walkstate.domain);
    SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

    var ntlsmd : bit;
    if IsFeatureImplemented(FEAT_LSMAOC) then
        case regime of
            when Regime\_EL30 => ntlsmd = SCTLR_S().nTLSMD;
            when Regime\_EL10 =>
                if HaveEL(EL3) && ELUsingAArch32(EL3) then
                    ntlsmd = SCTLR_NS().nTLSMD;
                else
                    ntlsmd = SCTLR().nTLSMD;
                end;
        end;
    else
        ntlsmd = '1';
    end;

    if AArch32\_S1HasAlignmentFaultDueToMemType(accdesc, aligned, ntlsmd, walkstate.memattrs) then
        fault.statuscode = Fault\_Alignment;
    elsif (! accdesc.acctype IN {AccessType\_IC, AccessType\_DC} &&
            domain == Domain\_NoAccess) then
        fault.statuscode = Fault\_Domain;
    elsif domain == Domain\_Client then
        if AArch32\_S1SDHasPermissionsFault(regime, walkstate.permissions,
                                           walkstate.memattrs.memtype,
                                           walkstate.baseaddress.paspace,
                                           accdesc) then
            fault.statuscode = Fault\_Permission;
        end;
    end;

    if fault.statuscode != Fault\_None then
        fault.domain = walkstate.domain;
        return (fault, ARBITRARY : AddressDescriptor, walkstate.sdftype);
    end;

    var memattrs : MemoryAttributes;
    if ((accdesc.acctype == AccessType\_IFETCH &&
        (walkstate.memattrs.memtype == MemType\_Device || !AArch32\_S1ICacheEnabled(regime))) ||
        (accdesc.acctype != AccessType\_IFETCH &&
        walkstate.memattrs.memtype == MemType\_Normal && !AArch32\_S1DCacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    else
        memattrs = walkstate.memattrs;
    end;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime\_EL10 && AArch32\_EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS\_Secure) then HCR().VM
         else HCR_EL2().VM) == '1' &&
        !(ImpDefBool("Apply effective shareability at stage 1"))) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);
    end;

    // Output Address
    let oa : FullAddress = AArch32\_SDStageOA(walkstate.baseaddress, va, walkstate.sdftype);
    let ipa : AddressDescriptor = CreateAddressDescriptor(ZeroExtend{64}(va),
                                                         oa, memattrs, accdesc);

```



```

    return (fault, ipa, walkstate.sdftype);
end;

```

Library pseudocode for aarch32/translation/translation/AArch32_S2Translate

```

// AArch32_S2Translate()
// =====
// Perform a stage 2 translation mapping an IPA to a PA

func AArch32_S2Translate(fault_in : FaultRecord, ipa : AddressDescriptor,
                        aligned : boolean, accdesc : AccessDescriptor
                        ) => (FaultRecord, AddressDescriptor)

begin

    var fault : FaultRecord = fault_in;
    assert IsZero(ipa.paddress.address[NUM\_PABITS-1:40]);
    if !ELStateUsingAArch32(EL2, accdesc.ss == SS\_Secure) then
        let slaarch64 : boolean = FALSE;
        return AArch64\_S2Translate(fault, ipa, slaarch64, aligned, accdesc);
    end;
    // Prepare fault fields in case a fault is detected
    fault.statuscode = Fault\_None;
    fault.secondstage = TRUE;
    fault.s2fslwalk = accdesc.acctype == AccessType\_TTW;
    fault.ipaddress = ipa.paddress;

    let walkparams : S2TTWParams = AArch32\_GetS2TTWParams();

    if walkparams.vm == '0' then
        // Stage 2 is disabled
        return (fault, ipa);
    end;

    if AArch32\_IPAIsOutOfRange(walkparams, ipa.paddress.address[39:0]) then
        fault.statuscode = Fault\_Translation;
        fault.level = 1;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var walkstate : TTWState;
    (fault, walkstate) = AArch32\_S2Walk(fault, walkparams, accdesc, ipa);

    if fault.statuscode != Fault\_None then
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if AArch32\_S2HasAlignmentFaultDueToMemType(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault\_Alignment;
    elsif AArch32\_S2HasPermissionsFault(walkparams,
                                        walkstate.permissions,
                                        walkstate.memattrs.memtype,
                                        accdesc) then
        fault.statuscode = Fault\_Permission;
    end;

    var s2_memattrs : MemoryAttributes;
    if ((accdesc.acctype == AccessType\_TTW &&
        walkstate.memattrs.memtype == MemType\_Device) ||
        (accdesc.acctype == AccessType\_IFETCH &&
        (walkstate.memattrs.memtype == MemType\_Device || HCR2().ID == '1')) ||
        (accdesc.acctype != AccessType\_IFETCH &&
        walkstate.memattrs.memtype == MemType\_Normal && HCR2().CD == '1')) then
        // Treat memory attributes as Normal Non-Cacheable
        s2_memattrs = NormalNCMemAttr();
        s2_memattrs.xs = walkstate.memattrs.xs;
    else
        s2_memattrs = walkstate.memattrs;
    end;

    let s2aarch64 : boolean = FALSE;
    let memattrs : MemoryAttributes = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
    let ipa_64 : bits(64) = ZeroExtend{}(ipa.paddress.address[39:0]);
    // Output Address
    let oa : FullAddress = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
    let pa : AddressDescriptor = CreateAddressDescriptor(ipa.vaddress, oa, memattrs, accdesc);

    return (fault, pa);
end;

```

Library pseudocode for aarch32/translation/translation/AArch32_SDStageOA

```
// AArch32_SDStageOA()
// =====
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address

func AArch32_SDStageOA(baseaddress : FullAddress, va : bits(32), sdftype : SDFTType) => FullAddress
begin
    let tsize : integer{} = SDFSIZE(sdftype);

    // Output Address
    var oa : FullAddress;
    oa.address = baseaddress.address[NUM\_PABITS-1:tsize] :: va[tsize-1:0];
    oa.paspace = baseaddress.paspace;
    return oa;
end;
```

Library pseudocode for aarch32/translation/translation/AArch32_TranslateAddress

```
// AArch32_TranslateAddress()
// =====
// Main entry point for translating an address

func AArch32_TranslateAddress(va : bits(32), accdesc : AccessDescriptor,
                             aligned : boolean, size : integer) => AddressDescriptor
begin
    let regime : Regime = TranslationRegime(PSTATE.EL);
    if !RegimeUsingAArch32(regime) then
        return AArch64\_TranslateAddress(ZeroExtend{64}(va), accdesc, aligned, size);
    end;

    var result : AddressDescriptor = AArch32\_FullTranslate(va, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType\_IFETCH then
        // For an instruction fetch, CheckDebug will be called
        // after the instruction is read from memory
        result.fault = AArch32\_CheckDebug(va, accdesc, size);
    end;

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend{64}(va);

    return result;
end;
```

Library pseudocode for aarch32/translation/translation/SDFSIZE

```
// SDFSIZE()
// =====
// Returns the short-descriptor format translation granule size

func SDFSIZE(sdftype : SDFTType) => AddressSize
begin
    case sdftype of
        when SDFTType\_SmallPage => return 12;
        when SDFTType\_LargePage => return 16;
        when SDFTType\_Section => return 20;
        when SDFTType\_Supersection => return 24;
        otherwise => unreachable;
    end;
end;
```

Library pseudocode for aarch32/translation/walk/AArch32_DecodeDescriptorTypeLD

```
// AArch32_DecodeDescriptorTypeLD()
// =====
// Determine whether the long-descriptor is a page, block or table

func AArch32_DecodeDescriptorTypeLD(descriptor : bits(64), level : integer) => DescriptorType
begin
    if descriptor[1:0] == '11' && level == FINAL\_LEVEL then
        return DescriptorType\_Leaf;
    elsif descriptor[1:0] == '11' then
        return DescriptorType\_Table;
    elsif descriptor[1:0] == '01' && level != FINAL\_LEVEL then
        return DescriptorType\_Leaf;
    else
        return DescriptorType\_Invalid;
    end;
end;
```

Library pseudocode for aarch32/translation/walk/AArch32_DecodeDescriptorTypeSD

```
// AArch32_DecodeDescriptorTypeSD()
// =====
// Determine the type of the short-descriptor

func AArch32_DecodeDescriptorTypeSD(descriptor : bits(32), level : integer) => SDFType
begin
    if level == 1 && descriptor[1:0] == '01' then
        return SDFType\_Table;
    elsif level == 1 && descriptor[18,1] == '01' then
        return SDFType\_Section;
    elsif level == 1 && descriptor[18,1] == '11' then
        return SDFType\_Supersection;
    elsif level == 2 && descriptor[1:0] == '01' then
        return SDFType\_LargePage;
    elsif level == 2 && descriptor[1:0] == '1x' then
        return SDFType\_SmallPage;
    else
        return SDFType\_Invalid;
    end;
end;
```

Library pseudocode for aarch32/translation/walk/AArch32_S1IASize

```
// AArch32_S1IASize()
// =====
// Retrieve the number of bits containing the input address for stage 1 translation

func AArch32_S1IASize(txsz : bits(3)) => AddressSize
begin
    return 32 - UInt(txsz);
end;
```



```

// AArch32_S1WalkLD()
// =====
// Traverse stage 1 translation tables in long format to obtain the final descriptor

func AArch32_S1WalkLD(fault_in : FaultRecord, regime : Regime,
    walkparams : S1TTWParams, accdesc : AccessDescriptor,
    va : bits(32)) => (FaultRecord, TTWState)

begin
    var fault : FaultRecord = fault_in;
    var txsz : bits(3);
    var ttbr : bits(64);
    var epd : bit;
    var varange : VARange;
    if regime == Regime\_EL2 then
        ttbr = HTTBR();
        txsz = walkparams.t0sz;
        varange = VARange\_LOWER;
    else
        varange = AArch32\_GetVARange(va, walkparams.t0sz, walkparams.t1sz);
        var ttbr0 : bits(64);
        var ttbr1 : bits(64);
        var ttbcr : TTBCR\_Type;
        if regime == Regime\_EL30 then
            ttbcr = TTBCR\_S();
            ttbr0 = TTBR0\_S();
            ttbr1 = TTBR1\_S();
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
            ttbcr = TTBCR\_NS();
            ttbr0 = TTBR0\_NS();
            ttbr1 = TTBR1\_NS();
        else
            ttbcr = TTBCR();
            ttbr0 = TTBR0();
            ttbr1 = TTBR1();
        end;

        assert ttbcr.EAE == '1';
        if varange == VARange\_LOWER then
            txsz = walkparams.t0sz;
            ttbr = ttbr0;
            epd = ttbcr.EPD0;
        else
            txsz = walkparams.t1sz;
            ttbr = ttbr1;
            epd = ttbcr.EPD1;
        end;
    end;

    if regime != Regime\_EL2 && epd == '1' then
        fault.level = 1;
        fault.statuscode = Fault\_Translation;
        return (fault, ARBITRARY : TTWState);
    end;

    // Input Address size
    let iasize : AddressSize = AArch32\_S1IASize(txsz);
    let granulebits : integer{} = TGxGranuleBits(walkparams.tgx) as integer{12, 14, 16};
    let stride : integer{} = granulebits - 3;
    let startlevel : integer = FINAL\_LEVEL - (((iasize-1) - granulebits) DIV stride);
    let levels : integer = FINAL\_LEVEL - startlevel;

    if !IsZero(ttbr[47:40]) then
        fault.statuscode = Fault\_AddressSize;
        fault.level = 0;
        return (fault, ARBITRARY : TTWState);
    end;

    var baseaddress : FullAddress;
    let baselsb = ((iasize - (levels*stride + granulebits)) + 3) as AddressSize;
    baseaddress.paspace = if accdesc.ss == SS\_Secure then PAS\_Secure else PAS\_NonSecure;
    baseaddress.address = ZeroExtend(NUM\_PABITS)(ttbr[39:baselsb]::Zeros{baselsb});

    var walkstate : TTWState;
    walkstate.baseaddress = baseaddress;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG = if HasUnprivileged(regime) then '1' else '0';
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    walkstate.permissions.ap_table = '00';
    walkstate.permissions.xn_table = '0';
    walkstate.permissions.pxn_table = '0';

    var descriptor : bits(64);
    var walkaddress : AddressDescriptor;

    walkaddress.vaddress = ZeroExtend{64}(va);

```

```

if !AArch32_S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;
end;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR().VM
     else HCR_EL2().VM) == '1' &&
    !(ImpDefBool("Apply effective shareability at stage 1")))) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);
end;

var desctype : DescriptorType;
var msb_residual : integer = iasize - 1;
repeat
    fault.level = walkstate.level;
    let indexlsb = ((FINAL_LEVEL - walkstate.level)*stride + granulebits) as AddressSize;
    let indexmsb = msb_residual as AddressSize;
    let index : bits(40) = ZeroExtend()(va[indexmsb:indexlsb]::'000');

    walkaddress.paddress.address = (walkstate.baseaddress.address OR
                                    ZeroExtend(NUM_PABITS)(index));
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    let toplevel : boolean = walkstate.level == startlevel;
    let walkaccess : AccessDescriptor = CreateAccDescS1TTW(toplevel, varange, accdesc);
    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) then
        let s2aligned : boolean = TRUE;
        let (s2fault, s2walkaddress) = AArch32_S2Translate(fault, walkaddress, s2aligned,
                                                            walkaccess);

        // Check for a fault on the stage 2 walk
        if s2fault.statuscode != Fault_None then
            return (s2fault, ARBITRARY : TTWState);
        end;

        (fault, descriptor) = FetchDescriptor{64}(walkparams.tee, s2walkaddress, walkaccess,
                                                  fault);
    else
        (fault, descriptor) = FetchDescriptor{64}(walkparams.tee, walkaddress, walkaccess,
                                                  fault);
    end;

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : TTWState);
    end;

    desctype = AArch32_DecodeDescriptorTypeLD(descriptor, walkstate.level);

case desctype of
    when DescriptorType_Table =>
        if !IsZero(descriptor[47:40]) then
            fault.statuscode = Fault_AddressSize;
            return (fault, ARBITRARY : TTWState);
        end;

        walkstate.baseaddress.address = ZeroExtend(NUM_PABITS)(descriptor[39:12]::
                                                                Zeros{12});
        if walkstate.baseaddress.paspace == PAS_Secure && descriptor[63] == '1' then
            walkstate.baseaddress.paspace = PAS_NonSecure;
        end;
        if walkparams.hpd == '0' then
            walkstate.permissions.xn_table = (walkstate.permissions.xn_table OR
                                              descriptor[60]);
            walkstate.permissions.ap_table = (walkstate.permissions.ap_table OR
                                              descriptor[62:61]);
            walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR
                                              descriptor[59]);
        end;
        walkstate.level = walkstate.level + 1;
        msb_residual = indexlsb - 1;

    when DescriptorType_Invalid =>
        fault.statuscode = Fault_Translation;
        return (fault, ARBITRARY : TTWState);

    when DescriptorType_Leaf =>
        walkstate.istable = FALSE;
end;

until desctype == DescriptorType_Leaf looplimit 3;

```

```

// Check the output address is inside the supported range
if !IsZero(descriptor[47:40]) then
    fault.statuscode = Fault AddressSize;
    return (fault, ARBITRARY : TTWState);
end;

// Check the access flag
if descriptor[10] == '0' then
    fault.statuscode = Fault AccessFlag;
    return (fault, ARBITRARY : TTWState);
end;

walkstate.permissions.xn = descriptor[54];
walkstate.permissions.pxn = descriptor[53];
walkstate.permissions.ap = descriptor[7:6]::'1';
walkstate.contiguous = descriptor[52];
if regime == Regime EL2 then
    // All EL2 regime accesses are treated as Global
    walkstate.nG = '0';
elseif accdesc.ss == SS Secure && walkstate.baseaddress.paspace == PAS NonSecure then
    // When a PE is using the Long-descriptor translation table format,
    // and is in Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk.
    walkstate.nG = '1';
else
    walkstate.nG = descriptor[11];
end;

let indexlsb = ((FINAL LEVEL - walkstate.level)*stride + granulebits) as AddressSize;
walkstate.baseaddress.address = ZeroExtend(NUM PABITS)(descriptor[39:indexlsb]::
    Zeros(indexlsb));
if walkstate.baseaddress.paspace == PAS Secure && descriptor[5] == '1' then
    walkstate.baseaddress.paspace = PAS NonSecure;
end;

let memattr : bits(3) = descriptor[4:2];
let sh : bits(2) = descriptor[9:8];
let attr : bits(8) = AArch32 MAIRAttr(UInt(memattr), walkparams.mair);
let slaarch64 : boolean = FALSE;
walkstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64);

return (fault, walkstate);
end;

```



```

// AArch32_S1WalkSD()
// =====
// Traverse stage 1 translation tables in short format to obtain the final descriptor

func AArch32_S1WalkSD(fault_in : FaultRecord, regime : Regime,
                      accdesc : AccessDescriptor, va : bits(32)) => (FaultRecord, TTWState)
begin
    var fault : FaultRecord = fault_in;
    var sctlr : SCTLR\_Type;
    var ttbcr : TTBCR\_Type;
    var ttbr0 : TTBR0\_Type;
    var ttbr1 : TTBR1\_Type;
    // Determine correct translation control registers to use.
    if regime == Regime\_EL30 then
        sctlr = SCTLR\_S\(\);
        ttbcr = TTBCR\_S\(\);
        ttbr0 = TTBR0\_S\(\);
        ttbr1 = TTBR1\_S\(\);
    elsif HaveEL\(EL3\) && ELUsingAArch32\(EL3\) then
        sctlr = SCTLR\_NS\(\);
        ttbcr = TTBCR\_NS\(\);
        ttbr0 = TTBR0\_NS\(\);
        ttbr1 = TTBR1\_NS\(\);
    else
        sctlr = SCTLR\(\);
        ttbcr = TTBCR\(\);
        ttbr0 = TTBR0\(\);
        ttbr1 = TTBR1\(\);
    end;

    assert ttbcr.EAE == '0';
    let ee : bit = sctlr.EE;
    let afe : bit = sctlr.AFE;
    let tre : bit = sctlr.TRE;
    let ttbcr_n : integer{} = UInt(ttbcr.N);
    let varange : VARange = (if ttbcr_n == 0 || IsZero(va[31:(32-ttbcr_n)]) then VARange\_LOWER
                             else VARange\_UPPER);
    let n : integer{} = if varange == VARange\_LOWER then ttbcr_n else 0;
    var ttb : bits(32);
    var pd : bits(1);
    var irgn : bits(2);
    var rgn : bits(2);
    var s : bits(1);
    var nos : bits(1);
    if varange == VARange\_LOWER then
        ttb = ttbr0.TTB0::Zeros{7};
        pd = ttbcr.PD0;
        irgn = ttbr0.IRGN;
        rgn = ttbr0.RGN;
        s = ttbr0.S;
        nos = ttbr0.NOS;
    else
        ttb = ttbr1.TTB1::Zeros{7};
        pd = ttbcr.PD1;
        irgn = ttbr1.IRGN;
        rgn = ttbr1.RGN;
        s = ttbr1.S;
        nos = ttbr1.NOS;
    end;

    // Check if Translation table walk disabled for translations with this Base register.
    if pd == '1' then
        fault.level = 1;
        fault.statuscode = Fault\_Translation;
        return (fault, ARBITRARY : TTWState);
    end;

    var baseaddress : FullAddress;
    baseaddress.paspace = if accdesc.ss == SS\_Secure then PAS\_Secure else PAS\_NonSecure;
    baseaddress.address = ZeroExtend{NUM\_PABITS}(ttb[31:14-n]::Zeros{14-n});

    let startlevel : integer = 1;
    var walkstate : TTWState;
    walkstate.baseaddress = baseaddress;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global. Translations in Short-Descriptor Format
    // always support global & non-global translations.
    walkstate.nG = '1';
    walkstate.memattrs = WalkMemAttrs(s::nos, irgn, rgn);
    walkstate.level = startlevel;
    walkstate.istable = TRUE;

    var domain : bits(4);
    var descriptor : bits(32);
    var walkaddress : AddressDescriptor;

    walkaddress.vaddress = ZeroExtend{64}(va);

```

```

if !AArch32_S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;
end;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR().VM
      else HCR_EL2().VM) == '1' &&
    !(ImpDefBool("Apply effective shareability at stage 1")))) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);
end;

var nG : bit;
var ns : bit;
var pxn : bit;
var ap : bits(3);
var tex : bits(3);
var c : bit;
var b : bit;
var xn : bit;
repeat
    fault.level = walkstate.level;

    var index : bits(32);
    if walkstate.level == 1 then
        index = ZeroExtend{32}(va[31-n:20]::'00');
    else
        index = ZeroExtend{32}(va[19:12]::'00');
    end;

    walkaddress.paddress.address = (walkstate.baseaddress.address OR
        ZeroExtend{NUM_PABITS}(index));
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    let toplevel : boolean = walkstate.level == startlevel;
    let walkaccess : AccessDescriptor = CreateAccDescS1TTW(toplevel, varange, accdesc);
    if regime == Regime_EL10 && AArch32_EL2Enabled(accdesc.ss) then
        let s2aligned : boolean = TRUE;
        let (s2fault, s2walkaddress) = AArch32_S2Translate(fault, walkaddress, s2aligned,
            walkaccess);

        if s2fault.statuscode != Fault_None then
            return (s2fault, ARBITRARY : TTWState);
        end;

        (fault, descriptor) = FetchDescriptor{32}(ee, s2walkaddress, walkaccess, fault);
    else
        (fault, descriptor) = FetchDescriptor{32}(ee, walkaddress, walkaccess, fault);
    end;

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : TTWState);
    end;

    walkstate.sdftype = AArch32_DecodeDescriptorTypeSD(descriptor, walkstate.level);

    case walkstate.sdftype of
        when SDFType_Invalid =>
            fault.domain = domain;
            fault.statuscode = Fault_Translation;
            return (fault, ARBITRARY : TTWState);

        when SDFType_Table =>
            domain = descriptor[8:5];
            ns = descriptor[3];
            pxn = descriptor[2];

            walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[31:10]::
                Zeros{10});
            walkstate.level = 2;

        when SDFType_SmallPage =>
            nG = descriptor[11];
            s = descriptor[10];
            ap = descriptor[9,5:4];
            tex = descriptor[8:6];
            c = descriptor[3];
            b = descriptor[2];
            xn = descriptor[0];

            walkstate.baseaddress.address = ZeroExtend{NUM_PABITS}(descriptor[31:12]::
                Zeros{12});
            walkstate.istable = FALSE;

```

```

when SDFType\_LargePage =>
    xn = descriptor[15];
    tex = descriptor[14:12];
    nG = descriptor[11];
    s = descriptor[10];
    ap = descriptor[9,5:4];
    c = descriptor[3];
    b = descriptor[2];

    walkstate.baseaddress.address = ZeroExtend(NUM\_PABITS)(descriptor[31:16]::
                                                Zeros{16});

    walkstate.istable = FALSE;

when SDFType\_Section =>
    ns = descriptor[19];
    nG = descriptor[17];
    s = descriptor[16];
    ap = descriptor[15,11:10];
    tex = descriptor[14:12];
    domain = descriptor[8:5];
    xn = descriptor[4];
    c = descriptor[3];
    b = descriptor[2];
    pxn = descriptor[0];

    walkstate.baseaddress.address = ZeroExtend(NUM\_PABITS)(descriptor[31:20]::
                                                Zeros{20});

    walkstate.istable = FALSE;

when SDFType\_Supersection =>
    ns = descriptor[19];
    nG = descriptor[17];
    s = descriptor[16];
    ap = descriptor[15,11:10];
    tex = descriptor[14:12];
    xn = descriptor[4];
    c = descriptor[3];
    b = descriptor[2];
    pxn = descriptor[0];
    domain = '0000';

    walkstate.baseaddress.address = ZeroExtend(NUM\_PABITS)(descriptor[8:5,23:20,31:24]::
                                                Zeros{24});

    walkstate.istable = FALSE;

end;

until walkstate.sdftype != SDFType\_Table looplimit 2;

if afe == '1' && ap[0] == '0' then
    fault.domain = domain;
    fault.statuscode = Fault\_AccessFlag;
    return (fault, ARBITRARY : TTWState);
end;

// Decode the TEX, C, B and S bits to produce target memory attributes
if tre == '1' then
    walkstate.memattrs = AArch32\_RemappedTEXDecode(regime, tex, c, b, s);
elsif RemapRegsHaveResetValues() then
    walkstate.memattrs = AArch32\_DefaultTEXDecode(tex, c, b, s);
else
    walkstate.memattrs = ImpDefMemoryAttributes("IMPLEMENTATION_DEFINED");
end;

walkstate.permissions.ap = ap;
walkstate.permissions.xn = xn;
walkstate.permissions.pxn = pxn;
walkstate.domain = domain;
walkstate.nG = nG;

if accdesc.ss == SS\_Secure && ns == '0' then
    walkstate.baseaddress.paspace = PAS\_Secure;
else
    walkstate.baseaddress.paspace = PAS\_NonSecure;
end;

return (fault, walkstate);
end;

```

Library pseudocode for aarch32/translation/walk/AArch32_S2IASize

```
// AArch32_S2IASize()
// =====
// Retrieve the number of bits containing the input address for stage 2 translation

func AArch32_S2IASize(t0sz : bits(4)) => AddressSize
begin
    return 32 - SInt(t0sz);
end;
```

Library pseudocode for aarch32/translation/walk/AArch32_S2StartLevel

```
// AArch32_S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

func AArch32_S2StartLevel(s10 : bits(2)) => integer
begin
    return 2 - UInt(s10);
end;
```



```

// AArch32_S2Walk()
// =====
// Traverse stage 2 translation tables in long format to obtain the final descriptor

func AArch32_S2Walk(fault_in : FaultRecord, walkparams : S2TTWParams,
                    accdesc : AccessDescriptor, ipa : AddressDescriptor) => (FaultRecord, TTWState)
begin
    var fault : FaultRecord = fault_in;

    if walkparams.sl0 == '1x' || AArch32\_S2InconsistentSL(walkparams) then
        fault.statuscode = Fault\_Translation;
        fault.level      = 1;
        return (fault, ARBITRARY : TTWState);
    end;

    // Input Address size
    let iasize : AddressSize = AArch32\_S2IASize(walkparams.t0sz);
    let startlevel : integer = AArch32\_S2StartLevel(walkparams.sl0);
    let levels : integer = FINAL\_LEVEL - startlevel;
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let stride : integer{} = granulebits - 3;

    if !IsZero(VTTBR()[47:40]) then
        fault.statuscode = Fault\_AddressSize;
        fault.level      = 0;
        return (fault, ARBITRARY : TTWState);
    end;

    var baseaddress : FullAddress;
    let baselsb = ((iasize - (levels*stride + granulebits)) + 3) as AddressSize;
    baseaddress.paspace = PAS\_NonSecure;
    baseaddress.address = ZeroExtend{NUM\_PABITS}(VTTBR()[39:baselsb]::Zeros{baselsb});

    var walkstate : TTWState;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                         walkparams.orgn);

    var descriptor : bits(64);
    let walkaccess : AccessDescriptor = CreateAccDescS2TTW(accdesc);
    var walkaddress : AddressDescriptor;

    walkaddress.vaddress = ipa.vaddress;

    if HCR2().CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;
    end;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    var msb_residual : integer = iasize - 1;
    var desctype : DescriptorType;
    repeat
        fault.level = walkstate.level;

        let indexlsb = ((FINAL\_LEVEL - walkstate.level)*stride + granulebits) as AddressSize;
        let indexmsb = msb_residual as AddressSize;
        let index : bits(40) = ZeroExtend{}(ipa.paddress.address[indexmsb:indexlsb]::'000');

        walkaddress.paddress.address = (walkstate.baseaddress.address OR
                                         ZeroExtend{NUM\_PABITS}(index));
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        (fault, descriptor) = FetchDescriptor{64}(walkparams.ee, walkaddress, walkaccess, fault);

        if fault.statuscode != Fault\_None then
            return (fault, ARBITRARY : TTWState);
        end;

        desctype = AArch32\_DecodeDescriptorTypeLD(descriptor, walkstate.level);

        case desctype of
            when DescriptorType\_Table =>
                if !IsZero(descriptor[47:40]) then
                    fault.statuscode = Fault\_AddressSize;
                    return (fault, ARBITRARY : TTWState);
                end;

                walkstate.baseaddress.address = ZeroExtend{NUM\_PABITS}(descriptor[39:12]::
                                                                    Zeros{12});
                walkstate.level = walkstate.level + 1;
                msb_residual = indexlsb - 1;

            when DescriptorType\_Invalid =>

```

```

        fault.statuscode = Fault Translation;
        return (fault, ARBITRARY : TTWState);

    when DescriptorType Leaf =>
        walkstate.istable = FALSE;
    end;

until desctype == DescriptorType Leaf looplimit 3;

// Check the output address is inside the supported range
if !IsZero(descriptor[47:40]) then
    fault.statuscode = Fault AddressSize;
    return (fault, ARBITRARY : TTWState);
end;

// Check the access flag
if descriptor[10] == '0' then
    fault.statuscode = Fault AccessFlag;
    return (fault, ARBITRARY : TTWState);
end;

// Unpack the descriptor into address and upper and lower block attributes
let indexlsb = ((FINAL LEVEL - walkstate.level)*stride + granulebits) as AddressSize;
walkstate.baseaddress.address = ZeroExtend(NUM PABITS)(descriptor[39:indexlsb]::
                                                Zeros{indexlsb});

walkstate.permissions.s2ap = descriptor[7:6];
walkstate.permissions.s2xn = descriptor[54];
if IsFeatureImplemented(FEAT_XNX) then
    walkstate.permissions.s2xnx = descriptor[53];
else
    walkstate.permissions.s2xnx = '0';
end;

let memattr : bits(4) = descriptor[5:2];
let sh : bits(2) = descriptor[9:8];
let s2aarch64 : boolean = FALSE;
walkstate.memattrs = S2DecodeMemAttrs(memattr, sh, s2aarch64);
walkstate.contiguous = descriptor[52];

return (fault, walkstate);
end;

```

Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

```

// RemapRegsHaveResetValues()
// =====

impdef func RemapRegsHaveResetValues() => boolean
begin
    return TRUE;
end;

```

Library pseudocode for aarch32/translation/walkparams/AArch32_GetS1TTWParams

```

// AArch32_GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

func AArch32_GetS1TTWParams(regime : Regime, va : bits(32)) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    case regime of
        when Regime EL2 => walkparams = AArch32\_S1TTWParamsEL2();
        when Regime EL10 => walkparams = AArch32\_S1TTWParamsEL10(va);
        when Regime EL30 => walkparams = AArch32\_S1TTWParamsEL30(va);
    end;

    return walkparams;
end;

```

Library pseudocode for aarch32/translation/walkparams/AArch32_GetS2TTWParams

```
// AArch32_GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

func AArch32_GetS2TTWParams() => S2TTWParams
begin
    var walkparams : S2TTWParams;

    walkparams.tgx = TGx\_4KB;
    walkparams.s = VTCR().S;
    walkparams.t0sz = VTCR().T0SZ;
    walkparams.sl0 = VTCR().SL0;
    walkparams.irgn = VTCR().IRGN0;
    walkparams.orgn = VTCR().ORGN0;
    walkparams.sh = VTCR().SH0;
    walkparams.ee = HSCTLR().EE;
    walkparams.ptw = HCR().PTW;
    walkparams.vm = HCR().VM OR HCR().DC;

    // VTCR.S must match VTCR.T0SZ[3]
    if walkparams.s != walkparams.t0sz[3] then
        (-, walkparams.t0sz) = ConstrainUnpredictableBits{4}(Unpredictable\_RESVTCRS);
    end;

    return walkparams;
end;
```

Library pseudocode for aarch32/translation/walkparams/AArch32_GetVARange

```
// AArch32_GetVARange()
// =====
// Select the translation base address for stage 1 long-descriptor walks

func AArch32_GetVARange(va : bits(32), t0sz : bits(3), t1sz : bits(3)) => VARange
begin
    // Lower range Input Address size
    let lo_iasize : AddressSize = AArch32\_S1IASize(t0sz);
    // Upper range Input Address size
    let up_iasize : AddressSize = AArch32\_S1IASize(t1sz);

    if t1sz == '000' && t0sz == '000' then
        return VARange\_LOWER;
    elsif t1sz == '000' then
        return if IsZero(va[31:lo_iasize]) then VARange\_LOWER else VARange\_UPPER;
    elsif t0sz == '000' then
        return if IsOnes(va[31:up_iasize]) then VARange\_UPPER else VARange\_LOWER;
    elsif IsZero(va[31:lo_iasize]) then
        return VARange\_LOWER;
    elsif IsOnes(va[31:up_iasize]) then
        return VARange\_UPPER;
    else
        // Will be reported as a Translation Fault
        return ARBITRARY : VARange;
    end;
end;
```

Library pseudocode for aarch32/translation/walkparams/AArch32_S1DCacheEnabled

```
// AArch32_S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

func AArch32_S1DCacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime\_EL30 => return SCTLR_S().C == '1';
        when Regime\_EL2 => return HSCTLR().C == '1';
        when Regime\_EL10 =>
            return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTLR_NS().C else SCTLR().C) == '1';
    end;
end;
```


Library pseudocode for aarch32/translation/walkparams/AArch32_S1ICacheEnabled

```
// AArch32_S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

func AArch32_S1ICacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime\_EL30 => return SCTLR_S().I == '1';
        when Regime\_EL2  => return HSCTLR().I == '1';
        when Regime\_EL10 =>
            return (if HaveEL\(EL3\) && ELUsingAArch32\(EL3\) then SCTLR_NS().I else SCTLR().I) == '1';
    end;
end;
```

Library pseudocode for aarch32/translation/walkparams/AArch32_S1TTWParamsEL10

```
// AArch32_S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

func AArch32_S1TTWParamsEL10(va : bits(32)) => S1TTWParams
begin
    var mair : bits(64);
    var sif : bit;
    var ttbcr : TTBCR_Type;
    var ttbcr2 : TTBCR2_Type;
    var sctlr : SCTLR_Type;

    if ELUsingAArch32\(EL3\) then
        ttbcr = TTBCR_NS();
        ttbcr2 = TTBCR2_NS();
        sctlr = SCTLR_NS();
        mair = MAIR1_NS()::MAIR0_NS();
        sif = SCR().SIF;
    else
        ttbcr = TTBCR();
        ttbcr2 = TTBCR2();
        sctlr = SCTLR();
        mair = MAIR1()::MAIR0();
        sif = if HaveEL\(EL3\) then SCR_EL3().SIF else '0';
    end;

    assert ttbcr.EAE == '1';
    var walkparams : S1TTWParams;

    walkparams.t0sz = ttbcr.T0SZ;
    walkparams.t1sz = ttbcr.T1SZ;
    walkparams.ee = sctlr.EE;
    walkparams.wxn = sctlr.WXN;
    walkparams.uwxn = sctlr.UWXN;
    walkparams.ntlsmid = if IsFeatureImplemented(FEAT_LSMAOC) then sctlr.NTLSMD else '1';
    walkparams.mair = mair;
    walkparams.sif = sif;

    let varange : VARange = AArch32\_GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange\_LOWER then
        walkparams.sh = ttbcr.SH0;
        walkparams.irgn = ttbcr.IRGN0;
        walkparams.orgn = ttbcr.ORGNO;
        walkparams.hpd = (if IsFeatureImplemented(FEAT_AA32HPD) then ttbcr.T2E AND ttbcr2.HPD0
            else '0');
    else
        walkparams.sh = ttbcr.SH1;
        walkparams.irgn = ttbcr.IRGN1;
        walkparams.orgn = ttbcr.ORGNI;
        walkparams.hpd = (if IsFeatureImplemented(FEAT_AA32HPD) then ttbcr.T2E AND ttbcr2.HPD1
            else '0');
    end;

    return walkparams;
end;
```

Library pseudocode for aarch32/translation/walkparams/AArch32_S1TTWParamsEL2

```
// AArch32_S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

func AArch32_S1TTWParamsEL2() => S1TTWParams
begin
    var walkparams : S1TTWParams;

    walkparams.tgx = TGx\_4KB;
    walkparams.t0sz = HTCR().T0SZ;
    walkparams.irgn = HTCR().IRGN0;
    walkparams.orgn = HTCR().ORGN0;
    walkparams.sh = HTCR().SH0;
    walkparams.hpd = if IsFeatureImplemented(FEAT_AA32HPD) then HTCR().HPD else '0';
    walkparams.ee = HSCTLR().EE;
    walkparams.wxn = HSCTLR().WXN;
    if IsFeatureImplemented(FEAT_LSMAOC) then
        walkparams.ntlsmid = HSCTLR().nTlSMD;
    else
        walkparams.ntlsmid = '1';
    end;

    walkparams.mair = HMAIR1()::HMAIR0();

    return walkparams;
end;
```

Library pseudocode for aarch32/translation/walkparams/AArch32_S1TTWParamsEL30

```
// AArch32_S1TTWParamsEL30()
// =====
// Gather stage 1 translation table walk parameters for EL3&0 regime

func AArch32_S1TTWParamsEL30(va : bits(32)) => S1TTWParams
begin
    assert TTBCR_S().EAE == '1';
    var walkparams : S1TTWParams;

    walkparams.t0sz = TTBCR_S().T0SZ;
    walkparams.tlsz = TTBCR_S().TlSZ;
    walkparams.ee = SCTLR_S().EE;
    walkparams.wxn = SCTLR_S().WXN;
    walkparams.uwxn = SCTLR_S().UWXN;
    walkparams.ntlsmid = if IsFeatureImplemented(FEAT_LSMAOC) then SCTLR_S().nTlSMD else '1';
    walkparams.mair = MAIR1_S()::MAIR0_S();
    walkparams.sif = SCR().SIF;

    let varange : VARange = AArch32\_GetVARange(va, walkparams.t0sz, walkparams.tlsz);
    if varange == VARange\_LOWER then
        walkparams.sh = TTBCR_S().SH0;
        walkparams.irgn = TTBCR_S().IRGN0;
        walkparams.orgn = TTBCR_S().ORGN0;
        walkparams.hpd = (if IsFeatureImplemented(FEAT_AA32HPD)
            then TTBCR_S().T2E AND TTBCR2_S().HPD0
            else '0');
    else
        walkparams.sh = TTBCR_S().SH1;
        walkparams.irgn = TTBCR_S().IRGN1;
        walkparams.orgn = TTBCR_S().ORGN1;
        walkparams.hpd = (if IsFeatureImplemented(FEAT_AA32HPD)
            then TTBCR_S().T2E AND TTBCR2_S().HPD1
            else '0');
    end;

    return walkparams;
end;
```

Library pseudocode for aarch64/debug/brbe

```
// Contents of the Branch Record Buffer
//=====

var Records_SRC : array [[64]] of BRBSRC_EL1_Type;
var Records_TGT : array [[64]] of BRBTGT_EL1_Type;
var Records_INF : array [[64]] of BRBINF_EL1_Type;

// Getter functions for branch records
//=====
// Functions used by MRS instructions that access branch records

func BRBSRC_EL1(n : integer) => BRBSRC_EL1_Type
begin
    assert n IN {0..31};
    let record_index : integer = UInt(BRBFCCR_EL1().BANK::n[4:0]);
    if record_index < GetBRBENumRecords() then
        return Records_SRC[[record_index]];
    else
        return Zeros{64};
    end;
end;

func BRBTGT_EL1(n : integer) => BRBTGT_EL1_Type
begin
    assert n IN {0..31};
    let record_index : integer = UInt(BRBFCCR_EL1().BANK::n[4:0]);
    if record_index < GetBRBENumRecords() then
        return Records_TGT[[record_index]];
    else
        return Zeros{64};
    end;
end;

func BRBINF_EL1(n : integer) => BRBINF_EL1_Type
begin
    assert n IN {0..31};
    let record_index : integer = UInt(BRBFCCR_EL1().BANK::n[4:0]);
    if record_index < GetBRBENumRecords() then
        return Records_INF[[record_index]];
    else
        return Zeros{64};
    end;
end;
```

Library pseudocode for aarch64/debug/brbe/BRBCycleCountingEnabled

```
// BRBCycleCountingEnabled()
// =====
// Returns TRUE if the recording of cycle counts is allowed,
// FALSE otherwise.

func BRBCycleCountingEnabled() => boolean
begin
    if HaveEL(EL2) && BRBCR_EL2().CC == '0' then return FALSE; end;
    if BRBCR_EL1().CC == '0' then return FALSE; end;
    return TRUE;
end;
```

Library pseudocode for aarch64/debug/brbe/BRBEBranch

```
// BRBEBranch()
// =====
// Called to write branch record for the following branches when BRBE is active:
// direct branches,
// indirect branches,
// direct branches with link,
// indirect branches with link,
// returns from subroutines.

func BRBEBranch(br_type : BranchType, cond : boolean, target_address : bits(64))
begin
    if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(br_type, cond) then
        var branch_type : bits(6);
        case br_type of
            when BranchType_DIR =>
                branch_type = if cond then '001000' else '000000';
            when BranchType_INDIR => branch_type = '000001';
            when BranchType_DIRCALL => branch_type = '000010';
            when BranchType_INDCALL => branch_type = '000011';
            when BranchType_RET => branch_type = '000101';
            otherwise => unreachable;
        end;

        var ccu : bit;
        var cc : bits(14);
        (ccu, cc) = BranchEncCycleCount();
        let el : bits(2) = PSTATE.EL;
        let mispredict : bit = (if BRBEMispredictAllowed() && BranchMispredict() then '1'
                                else '0');

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                '11', PC64(), target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;

    return;
end;
```

Library pseudocode for aarch64/debug/brbe/BRBEBranchOnISB

```
// BRBEBranchOnISB()
// =====
// Returns TRUE if ISBs generate Branch records, and FALSE otherwise.

func BRBEBranchOnISB() => boolean
begin
    return ImpDefBool("ISB generates Branch records");
end;
```

Library pseudocode for aarch64/debug/brbe/BRBEDebugStateEntry

```
// BRBEDebugStateEntry()
// =====
// Called to write Debug state entry branch record when BRBE is active.

func BRBEDebugStateEntry(source_address : bits(64))
begin
    if BranchRecordAllowed(PSTATE.EL) then
        let branch_type : bits(6) = '100001';
        var ccu : bit;
        var cc : bits(14);
        (ccu, cc) = BranchEncCycleCount();
        let el : bits(2) = '00';
        let mispredict : bit = '0';

        // Debug state is a prohibited region, therefore target_address=0
        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                '10', source_address, Zeros{64});

        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;

    return;
end;
```

Library pseudocode for aarch64/debug/brbe/BRBEDebugStateExit

```
// BRBEDebugStateExit()
// =====
// Called to write Debug state exit branch record when BRBE is active.

func BRBEDebugStateExit(target_address : bits(64))
begin
    if BranchRecordAllowed(PSTATE.EL) then
        // Debug state is a prohibited region, therefore ccu=1, cc=0, source_address=0
        let branch_type : bits(6) = '111001';
        let ccu : bit = '1';
        let cc : bits(14) = Zeros{14};
        let el : bits(2) = PSTATE.EL;
        let mispredict : bit = '0';

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                '01', Zeros{64}, target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;

    return;
end;
```



```

// BRBEEException()
// =====
// Called to write exception branch record when BRBE is active.

func BRBEEException(erec : ExceptionRecord, source_valid : boolean,
    source_address_in : bits(64),
    target_address_in : bits(64), target_el : bits(2),
    trappedsyscallinst : boolean)

begin
    var target_address : bits(64) = target_address_in;
    let except : Exception = erec.exceptype;
    let iss : bits(25) = erec.syndrome.iss;
    case target_el of
        when EL3 =>
            if !IsFeatureImplemented(FEAT_BRBEv1p1) || (MDCR_EL3().E3BREC == MDCR_EL3().E3BREW) then
                return;
            end;
        when EL2 => if BRBCR_EL2().EXCEPTION == '0' then return; end;
        when EL1 => if BRBCR_EL1().EXCEPTION == '0' then return; end;
    end;

    let target_valid : boolean = BranchRecordAllowed(target_el);

    if source_valid || target_valid then
        var branch_type : bits(6);
        case except of
            when Exception Uncategorized => branch_type = '100011'; // Trap
            when Exception WFXTrap => branch_type = '100011'; // Trap
            when Exception CP15RTTTrap => branch_type = '100011'; // Trap
            when Exception CP15RRTTrap => branch_type = '100011'; // Trap
            when Exception CP14RRTTrap => branch_type = '100011'; // Trap
            when Exception CP14DTTTrap => branch_type = '100011'; // Trap
            when Exception AdvSIMDFPAccessTrap => branch_type = '100011'; // Trap
            when Exception FPIDTrap => branch_type = '100011'; // Trap
            when Exception PACTrap => branch_type = '100011'; // Trap
            when Exception CP14RRTTrap => branch_type = '100011'; // Trap
            when Exception OtherTrap => branch_type = '100011'; // Trap
            when Exception BranchTarget => branch_type = '101011'; // Inst Fault
            when Exception IllegalState => branch_type = '100011'; // Trap
            when Exception SupervisorCall =>
                if !trappedsyscallinst then
                    branch_type = '100010'; // Call
                else
                    branch_type = '100011'; // Trap
                end;
            when Exception HypervisorCall => branch_type = '100010'; // Call
            when Exception MonitorCall =>
                if !trappedsyscallinst then
                    branch_type = '100010'; // Call
                else
                    branch_type = '100011'; // Trap
                end;
            when Exception SystemRegisterTrap => branch_type = '100011'; // Trap
            when Exception SystemRegister128Trap => branch_type = '100011'; // Trap
            when Exception SVEAccessTrap => branch_type = '100011'; // Trap
            when Exception SMEAccessTrap => branch_type = '100011'; // Trap
            when Exception ERETTrap => branch_type = '100011'; // Trap
            when Exception PACFail => branch_type = '101100'; // Data Fault
            when Exception InstructionAbort => branch_type = '101011'; // Inst Fault
            when Exception PCAlignment => branch_type = '101010'; // Alignment
            when Exception DataAbort => branch_type = '101100'; // Data Fault
            when Exception NV2DataAbort => branch_type = '101100'; // Data Fault
            when Exception SPAlignment => branch_type = '101010'; // Alignment
            when Exception FPTrappedException => branch_type = '100011'; // Trap
            when Exception SError => branch_type = '100100'; // System Error
            when Exception Breakpoint => branch_type = '100110'; // Inst debug
            when Exception SoftwareStep => branch_type = '100110'; // Inst debug
            when Exception Watchpoint => branch_type = '100111'; // Data debug
            when Exception NV2Watchpoint => branch_type = '100111'; // Data debug
            when Exception SoftwareBreakpoint => branch_type = '100110'; // Inst debug
            when Exception IRQ => branch_type = '101110'; // IRQ
            when Exception FIQ => branch_type = '101111'; // FIQ
            when Exception MemCpyMemSet => branch_type = '100011'; // Trap
            when Exception GCSFail =>
                if iss[23:20] == '0000' then
                    branch_type = '101100'; // Data Fault
                elsif iss[23:20] == '0001' then
                    branch_type = '101011'; // Inst Fault
                elsif iss[23:20] == '0010' then
                    branch_type = '100011'; // Trap
                else
                    unreachable;
                end;
            when Exception Profiling => branch_type = '100110'; // Inst debug
            when Exception GPC =>
                if iss[20] == '1' then
                    branch_type = '101011'; // Inst fault
                else
                    branch_type = '101100'; // Data fault
                end;
            otherwise => unreachable;
        end;

        var ccu : bit;
        var cc : bits(14);
        (ccu, cc) = BranchEncCycleCount();
        let el : bits(2) = if target_valid then target_el else '00';
        let mispredict : bit = '0';
        let sv : bit = if source_valid then '1' else '0';
    end;
end;

```

```

    let tv : bit = if target_valid then '1' else '0';
    let source_address : bits(64) = if source_valid then source_address_in else Zeros{64};

    if !target_valid then
        target_address = Zeros{64};
    else
        target_address = AArch64\_BranchAddr(target_address, target_el);
    end;

    UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                               sv::tv, source_address, target_address);

    PMUEvent(PMU_EVENT_BRB_FILTRATE);
end;

return;
end;

```

Library pseudocode for aarch64/debug/brbe/BRBEEExceptionReturn

```

// BRBEEExceptionReturn()
// =====
// Called to write exception return branch record when BRBE is active.

func BRBEEExceptionReturn(target_address_in : bits(64), source_el : bits(2),
                           source_valid : boolean, source_address_in : bits(64))
begin
    var target_address : bits(64) = target_address_in;
    case source_el of
        when EL3 =>
            if !IsFeatureImplemented(FEAT_BRBEv1p1) || (MDCR_EL3().E3BREC == MDCR_EL3().E3BREW) then
                return;
            end;
        when EL2 => if BRBCR_EL2().ERTN == '0' then return; end;
        when EL1 => if BRBCR_EL1().ERTN == '0' then return; end;
    end;

    let target_valid : boolean = BranchRecordAllowed(PSTATE.EL);

    if source_valid || target_valid then
        let branch_type : bits(6) = '000111';
        var ccu : bit;
        var cc : bits(14);
        (ccu, cc) = BranchEncCycleCount();
        let el : bits(2) = if target_valid then PSTATE.EL else '00';
        let mispredict : bit = (if source_valid && BRBEMispredictAllowed() &&
                                BranchMispredict() then '1' else '0');
        let sv : bit = if source_valid then '1' else '0';
        let tv : bit = if target_valid then '1' else '0';
        let source_address : bits(64) = if source_valid then source_address_in else Zeros{64};
        if !target_valid then
            target_address = Zeros{64};
        end;

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                sv::tv, source_address, target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;

    return;
end;

```

Library pseudocode for aarch64/debug/brbe/BRBEFreeze

```

// BRBEFreeze()
// =====
// Generates BRBE freeze event.

func BRBEFreeze()
begin
    BRBFCR_EL1().PAUSED = '1';
    BRBTS_EL1() = GetTimestamp(BRBETimeStamp());
end;

```

Library pseudocode for aarch64/debug/brbe/BRBEISB

```

// BRBEISB()
// =====
// Handles ISB instruction for BRBE.

func BRBEISB()
begin
    let branch_conditional : boolean = FALSE;
    BRBEBranch(BranchType\_DIR, branch_conditional, PC64() + 4);
end;

```


Library pseudocode for aarch64/debug/brbe/BRBEMispredictAllowed

```
// BRBEMispredictAllowed()
// =====
// Returns TRUE if the recording of branch misprediction is allowed,
// FALSE otherwise.

func BRBEMispredictAllowed() => boolean
begin
    if HaveEL(EL2) && BRBCR_EL2().MPRED == '0' then return FALSE; end;
    if BRBCR_EL1().MPRED == '0' then return FALSE; end;
    return TRUE;
end;
```

Library pseudocode for aarch64/debug/brbe/BRBETimeStamp

```
// BRBETimeStamp()
// =====
// Returns captured timestamp.

func BRBETimeStamp() => TimeStamp
begin
    if HaveEL(EL2) then
        var TS_el2 : bits(2) = BRBCR_EL2().TS;
        if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits{2}(Unpredictable\_EL2TIMESTAMP);
        end;
        case TS_el2 of
            when '00' =>
                // Falls out to check BRBCR_EL1.TS
                pass;
            when '01' =>
                return TimeStamp Virtual;
            when '10' =>
                assert IsFeatureImplemented(FEAT_ECV); // Otherwise ConstrainUnpredictableBits
                // removes this case
                return TimeStamp OffsetPhysical;
            when '11' =>
                return TimeStamp Physical;
        end;
    end;

    var TS_el1 : bits(2) = BRBCR_EL1().TS;
    if TS_el1 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_el1 == '10') then
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits{2}(Unpredictable\_EL1TIMESTAMP);
    end;
    case TS_el1 of
        when '01' =>
            return TimeStamp Virtual;
        when '10' =>
            return TimeStamp OffsetPhysical;
        when '11' =>
            return TimeStamp Physical;
        otherwise =>
            unreachable; // ConstrainUnpredictableBits removes this case
    end;
end;
```

Library pseudocode for aarch64/debug/brbe/BRB_IALL

```
// BRB_IALL()
// =====
// Called to perform invalidation of branch records

func BRB_IALL()
begin
    let nbrberecords : integer = GetBRBNumRecords();
    for i = 0 to nbrberecords - 1 do
        Records\_SRC[[i]] = Zeros{64};
        Records\_TGT[[i]] = Zeros{64};
        Records\_INF[[i]] = Zeros{64};
    end;
end;
```

Library pseudocode for aarch64/debug/brbe/BRB_INJ

```
// BRB_INJ()
// =====
// Called to perform manual injection of branch records.

func BRB_INJ()
begin
    UpdateBranchRecordBuffer(BRBINFINJ_EL1().CCU, BRBINFINJ_EL1().CC,
                             BRBINFINJ_EL1().TYPE, BRBINFINJ_EL1().EL,
                             BRBINFINJ_EL1().MPRED, BRBINFINJ_EL1().VALID, BRBSRCINJ_EL1().ADDRESS,
                             BRBTGTINJ_EL1().ADDRESS);
    BRBINFINJ_EL1() = ARBITRARY : bits(64);
    BRBSRCINJ_EL1() = ARBITRARY : bits(64);
    BRBTGTINJ_EL1() = ARBITRARY : bits(64);

    if ConstrainUnpredictableBool(Unpredictable\_BRBFILTRATE) then
        PMUEvent(PMU_EVENT_BRB_FILTRATE);
    end;
end;
```

Library pseudocode for aarch64/debug/brbe/BranchEncCycleCount

```
// BranchEncCycleCount()
// =====

// The first return result is '1' if either of the following is true, and '0' otherwise:
// - This is the first Branch record after the PE exited a Prohibited Region.
// - This is the first Branch record after cycle counting has been enabled.
// If the first return is '0', the second return result is the encoded cycle count
// since the last branch.
// The format of this field uses a mantissa and exponent to express the cycle count value.
// - bits[7:0] indicate the mantissa M.
// - bits[13:8] indicate the exponent E.
// The cycle count is expressed using the following function:
// cycle_count = (if IsZero(E) then UInt(M) else UInt('1'::M::Zeros(UInt(E)-1)))
// A value of all ones in both the mantissa and exponent indicates the cycle count value
// exceeded the size of the cycle counter.
// If the cycle count is not known, the second return result is zero.

func BranchEncCycleCount() => (bit, bits(14))
begin
    var cc : integer = BranchRawCycleCount();
    if BRBCycleCountingEnabled() || FirstBranchAfterProhibited() then
        return ('1', Zeros{14});
    end;

    // The format of this field uses a mantissa and exponent to express the cycle count value.
    // - bits[7:0] indicate the mantissa M.
    // - bits[13:8] indicate the exponent E.
    // The cycle count is expressed using the following function:
    // cycle_count = (if IsZero(E) then UInt(M) else UInt('1'::M::Zeros(UInt(E)-1)))
    // A value of all ones in both the mantissa and exponent indicates the cycle count value
    // exceeded the size of the cycle counter.

    var E : bits(6);
    var M : bits(8);

    if cc < 2^8 then
        E = Zeros{6};
        M = cc[7:0];
    elsif cc >= 2^20 then
        E = Ones{6};
        M = Ones{8};
    else
        E = 1[5:0];
        while cc >= 2^9 looplimit 11 do
            E = E + 1;
            cc = cc DIVRM 2;
        end;
        M = cc[7:0];
    end;

    return ('0', E::M);
end;
```

Library pseudocode for aarch64/debug/brbe/BranchMispredict

```
// BranchMispredict()
// =====
// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.

impdef func BranchMispredict() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for aarch64/debug/brbe/BranchRawCycleCount

```
// BranchRawCycleCount()
// =====
// If the cycle count is known, the return result is the cycle count since the last branch.

impdef func BranchRawCycleCount() => integer
begin
    return 0;
end;
```

Library pseudocode for aarch64/debug/brbe/BranchRecordAllowed

```
// BranchRecordAllowed()
// =====
// Returns TRUE if branch recording is allowed, FALSE otherwise.

func BranchRecordAllowed(el : bits(2)) => boolean
begin
    if ELUsingAArch32(el) then
        return FALSE;
    end;

    if BRBFCCR_EL1().PAUSED == '1' then
        return FALSE;
    end;

    if el == EL3 && IsFeatureImplemented(FEAT_BRBEv1p1) then
        return (MDCR_EL3().E3BREC != MDCR_EL3().E3BREW);
    end;

    if HaveEL(EL3) && (MDCR_EL3().SBRBE == '00' ||
        (CurrentSecurityState() == SS\_Secure && MDCR_EL3().SBRBE == '01')) then
        return FALSE;
    end;

    case el of
        when EL3 => return FALSE; // FEAT_BRBEv1p1 not implemented
        when EL2 => return BRBCCR_EL2().E2BRE == '1';
        when EL1 => return BRBCCR_EL1().E1BRE == '1';
        when EL0 =>
            if EL2Enabled() && HCR_EL2().TGE == '1' then
                return BRBCCR_EL2().E0HBRE == '1';
            else
                return BRBCCR_EL1().E0BRE == '1';
            end;
    end;
end;
```

Library pseudocode for aarch64/debug/brbe/FilterBranchRecord

```
// FilterBranchRecord()
// =====
// Returns TRUE if the branch record is not filtered out, FALSE otherwise.

func FilterBranchRecord(br : BranchType, cond : boolean) => boolean
begin
    case br of
        when BranchType\_DIRCALL =>
            return BRBFCCR_EL1().DIRCALL != BRBFCCR_EL1().EnI;
        when BranchType\_INDCALL =>
            return BRBFCCR_EL1().INDCALL != BRBFCCR_EL1().EnI;
        when BranchType\_RET =>
            return BRBFCCR_EL1().RTN != BRBFCCR_EL1().EnI;
        when BranchType\_DIR =>
            if cond then
                return BRBFCCR_EL1().CONDDIR != BRBFCCR_EL1().EnI;
            else
                return BRBFCCR_EL1().DIRECT != BRBFCCR_EL1().EnI;
            end;
        when BranchType\_INDIR =>
            return BRBFCCR_EL1().INDIRECT != BRBFCCR_EL1().EnI;
        otherwise => unreachable;
    end;
    return FALSE;
end;
```

Library pseudocode for aarch64/debug/brbe/FirstBranchAfterProhibited

```
// FirstBranchAfterProhibited()
// =====
// Returns TRUE if branch recorded is the first branch after a prohibited region,
// FALSE otherwise.

impdef func FirstBranchAfterProhibited() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for aarch64/debug/brbe/GetBRBNumRecords

```
// GetBRBNumRecords()
// =====
// Returns the number of branch records implemented.

func GetBRBNumRecords() => integer
begin
    assert UInt(BRBIDR0_EL1().NUMREC) IN {0x08, 0x10, 0x20, 0x40};
    return ImpDefInt("Number of BRBE Branch records");
end;
```

Library pseudocode for aarch64/debug/brbe/ShouldBRBEFreeze

```
// ShouldBRBEFreeze()
// =====
// Returns TRUE if the BRBE freeze event conditions have been met, and FALSE otherwise.

func ShouldBRBEFreeze() => boolean
begin
    if !BranchRecordAllowed(PSTATE.EL) then
        return FALSE;
    end;

    var include_r1 : boolean;
    var include_r2 : boolean;
    let include_r3 : boolean = FALSE;

    if HaveEL(EL2) then
        include_r1 = (BRBCR_EL1().FZP == '1');
        include_r2 = (BRBCR_EL2().FZP == '1');
    else
        include_r1 = TRUE;
        include_r2 = TRUE;
    end;

    return CheckPMUOverflowCondition(PMUOverflowCondition BRBEFreeze,
                                   include_r1, include_r2, include_r3);
end;
```

Library pseudocode for aarch64/debug/brbe/UpdateBranchRecordBuffer

```
// UpdateBranchRecordBuffer()
// =====
// Add a new Branch record to the buffer.

func UpdateBranchRecordBuffer(ccu : bit, cc : bits(14), branch_type : bits(6), el : bits(2),
                             mispredict : bit, valid : bits(2),
                             source_address : bits(64), target_address : bits(64))
begin
    // Shift the Branch Records in the buffer
    let nrberecords : integer = GetBRBNumRecords();
    for i = nrberecords - 1 downto 1 do
        Records SRC[[i]] = Records SRC[[i - 1]];
        Records TGT[[i]] = Records TGT[[i - 1]];
        Records INF[[i]] = Records INF[[i - 1]];
    end;

    Records INF[[0]].CCU = ccu;
    Records INF[[0]].CC = cc;

    Records INF[[0]].EL = el;
    Records INF[[0]].VALID = valid;
    Records INF[[0]].MPRED = mispredict;
    Records INF[[0]].TYPE = branch_type;

    Records SRC[[0]] = source_address;
    Records TGT[[0]] = target_address;

    return;
end;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64_BreakpointMatch

```
// AArch64_BreakpointMatch()
// =====
// Breakpoint matching in an AArch64 translation regime.
// Returns BreakpointInfo structure that contains breakpoint type, a boolean to indicate the
// type of breakpoint, matched address and whether the breakpoint is active and matched
// successfully. For Address Mismatch breakpoints, the returned boolean is the inverted result.
func AArch64_BreakpointMatch(n : integer, vaddress : bits(64), accdesc : AccessDescriptor,
                             size : integer) => BreakpointInfo
begin
    assert !ELUsingAArch32(S1TranslationRegime() );
    assert n < NumBreakpointsImplemented();
    var brkptinfo : BreakpointInfo;

    let linking_enabled : boolean = (DBGBCR_EL1(n).BT IN {'0x11', '1xx1'}) ||
                                     (IsFeatureImplemented(FEAT_ABLE) && DBGBCR_EL1(n).BT2 == '1');

    // A breakpoint that has linking enabled does not generate debug events in isolation
    if linking_enabled then
        brkptinfo.bptype = BreakpointType\_Inactive;
        brkptinfo.match = FALSE;
        return brkptinfo;
    end;

    let enabled : boolean      = IsBreakpointEnabled(n);
    let linked : boolean       = DBGBCR_EL1(n).BT == '0x01';
    let isbreakpnt : boolean   = TRUE;
    let linked_to : boolean     = FALSE;
    let from_linking_enabled : boolean = FALSE;
    let lbnx : bits(2) = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGBCR_EL1(n).LBNX else '00';
    let linked_n : integer{} = UInt(lbnx :: DBGBCR_EL1(n).LBN);
    let ssce : bit = if IsFeatureImplemented(FEAT_RME) then DBGBCR_EL1(n).SSCE else '0';
    let state_match = AArch64\_StateMatch(DBGBCR_EL1(n).SSC, ssce, DBGBCR_EL1(n).HMC,
                                         DBGBCR_EL1(n).PMC, linked, linked_n, isbreakpnt,
                                         vaddress, accdesc);

    var (bp_type, value_match) = AArch64\_BreakpointValueMatch(n, vaddress, linked_to, isbreakpnt,
                                                             from_linking_enabled);

    if HaveAArch32() && size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        let (-, match_i) = AArch64\_BreakpointValueMatch(n, vaddress + 2, linked_to, isbreakpnt,
                                                         from_linking_enabled);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);
        end;
    end;

    if vaddress[1] == '1' && DBGBCR_EL1(n).BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1(n).BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR_EL1(n)+2.
        if value_match then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);
        end;
    end;

    if !(state_match && enabled) then
        brkptinfo.bptype = BreakpointType\_Inactive;
        brkptinfo.match = FALSE;
    else
        brkptinfo.bptype = bp_type;
        brkptinfo.match = value_match;
    end;
    return brkptinfo;
end;
```



```

// AArch64_BreakpointValueMatch()
// =====
// Returns breakpoint type to indicate the type of breakpoint and a boolean to indicate
// whether the breakpoint matched successfully. For Address Mismatch breakpoints, the
// returned boolean is the inverted result. If the breakpoint type return value is Inactive,
// then the boolean result is FALSE.

func AArch64_BreakpointValueMatch(n_in : integer, vaddress : bits(64),
                                  linked_to : boolean, isbreakpnt : boolean,
                                  from_linking_enabled : boolean
                                  ) => (BreakpointType, boolean) recurselimit 2

begin

    // "n_in" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    // matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    // "isbreakpnt" TRUE is this is a call from BreakpointMatch or from StateMatch for a
    // linked breakpoint or from BreakpointValueMatch for a linked breakpoint with linking enabled.
    // "from_linking_enabled" is TRUE if this is a call from BreakpointValueMatch for a linked
    // breakpoint with linking enabled.
    var n : integer = n_in;
    var c : Constraint;
    var dbgtype : bits(5);

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                              Unpredictable_BPNOTIMPL);
        assert c IN {Constraint DISABLED, Constraint UNKNOWN};
        if c == Constraint DISABLED then return (BreakpointType Inactive, FALSE); end;
    end;

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if !IsBreakpointEnabled(n) then return (BreakpointType Inactive, FALSE); end;

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    if IsFeatureImplemented(FEAT_ABLE) then
        dbgtype = DBGBCR_EL1(n).[BT2,BT];
    else
        dbgtype = '0' :: DBGBCR_EL1(n).BT;
    end;

    (c, dbgtype) = AArch64_ReservedBreakpointType(n, dbgtype);
    if c == Constraint DISABLED then return (BreakpointType Inactive, FALSE); end;
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    let match_addr : boolean = (dbgtype == 'x0x0x');
    let mismatch : boolean = (dbgtype == 'x010x');
    let match_vmid : boolean = (dbgtype == 'x10xx');
    let match_cid : boolean = (dbgtype == 'x001x');
    let match_cid1 : boolean = (dbgtype IN {'x101x', 'xx11x'});
    let match_cid2 : boolean = (dbgtype == 'x11xx');
    let linking_enabled : boolean = (dbgtype IN {'xxx11', 'x1xx1', '1xxxx'});

    // If this is a call from StateMatch, return FALSE if the breakpoint is not
    // programmed with linking enabled.
    if linked_to && !linking_enabled then
        return (BreakpointType Inactive, FALSE);
    end;

    // If called from BreakpointMatch return FALSE for breakpoint with linking enabled.
    if !linked_to && linking_enabled then
        return (BreakpointType Inactive, FALSE);
    end;

    let linked : boolean = (dbgtype == 'x0x01');
    if from_linking_enabled then // A breakpoint with linking enabled has called this function.
        assert linked_to && isbreakpnt;
        if linked then
            // A breakpoint with linking enabled is linked to a linked breakpoint. This is
            // architecturally UNPREDICTABLE, but treated as disabled in the pseudo code to
            // avoid potential recursion in BreakpointValueMatch().
            return (BreakpointType Inactive, FALSE);
        end;
    end;

    // If a linked breakpoint is linked to an address matching breakpoint,
    // the behavior is CONSTRAINED UNPREDICTABLE.
    if linked_to && match_addr && isbreakpnt then
        if !ConstrainUnpredictableBool(Unpredictable_BPLINKEDADDRMATCH) then
            return (BreakpointType Inactive, FALSE);
        end;
    end;

    // A breakpoint programmed for address mismatch does not match in AArch32 state.

```

```

if mismatch && UsingAArch32() then
    return (BreakpointType_Inactive, FALSE);
end;

var bvr_match : boolean = FALSE;
var bxvr_match : boolean = FALSE;
var bp_type : BreakpointType;
var mask : integer{0..31};

if IsFeatureImplemented(FEAT_BWE) then
    mask = UInt(DBGBCR_EL1(n).MASK);

    // If the mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask IN {1, 2} then
        var unpred_mask : integer;
        (c, unpred_mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESBPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED => return (BreakpointType_Inactive, FALSE); // Disabled
            when Constraint_NONE => mask = 0; // No masking
            // Otherwise the value returned by ConstrainUnpredictableBits must
            // be a not-reserved value.
            otherwise => mask = unpred_mask as integer{3..31};
        end;
    end;

    if mask != 0 then
        // When DBGBCR_EL1(n).MASK is a valid nonzero value, the behavior is
        // CONSTRAINED UNPREDICTABLE if any of the following are true:
        // - DBGBCR_EL1(n).[BT2,BT] is programmed for a Context matching breakpoint.
        // - DBGBCR_EL1(n).BAS is not '1111' and AArch32 is supported at EL0.
        if ((match_cid || match_cid1 || match_cid2) ||
            (DBGBCR_EL1(n).BAS != '1111' && HaveAArch32())) then
            if !ConstrainUnpredictableBool(Unpredictable_BPMASK) then
                return (BreakpointType_Inactive, FALSE);
            end;
        end;
    else
        // A stand-alone mismatch of a single address is not supported.
        if mismatch then
            return (BreakpointType_Inactive, FALSE);
        end;
    end;

else
    mask = 0;
end;

// Do the comparison.
if match_addr then
    var byte_select_match : boolean;
    let byte : integer = UInt(vaddress[1:0]);

    if HaveAArch32() then
        // T32 instructions can be executed at EL0 in an AArch64 translation regime.
        assert byte IN {0,2}; // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR_EL1(n).BAS[byte] == '1');
    else
        assert byte == 0; // "vaddress" is word aligned
        byte_select_match = TRUE; // DBGBCR_EL1(n).BAS[byte] is RES1
    end;

    // When FEAT_LVA3 is not implemented, if the DBGBCR_EL1(n).RESS field bits are not a
    // sign extension of the MSB of DBGBCR_EL1(n).VA, it is UNPREDICTABLE whether they
    // appear to be included in the match.
    // If 'vaddress' is outside of the current virtual address space, then the access
    // generates a Translation fault.
    let dbgtop : AddressSize = DebugAddrTop();
    let unpredictable_ress : boolean = (dbgtop < 55 && !IsOnes(DBGBCR_EL1(n)[63:dbgtop]) &&
        !IsZero(DBGBCR_EL1(n)[63:dbgtop]) &&
        ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS));
    let cmpmsb : integer{} = if unpredictable_ress then 63 else dbgtop;
    let cmpls : integer{} = if mask > 2 then mask else 2;
    bvr_match = ((vaddress[cmpmsb:cmpls] == DBGBCR_EL1(n)[cmpmsb:cmpls]) &&
        byte_select_match);
    if mask > 2 then
        // If masked bits of DBGBCR_EL1(n) are not zero, the behavior
        // is CONSTRAINED UNPREDICTABLE.
        let masktop : integer{} = mask - 1;
        if bvr_match && !IsZero(DBGBCR_EL1(n)[masktop:2]) then
            bvr_match = ConstrainUnpredictableBool(Unpredictable_BPMASKEDBITS);
        end;
    end;

elseif match_cid then
    if IsInHost() then
        bvr_match = (CONTEXTIDR_EL2()[31:0] == DBGBCR_EL1(n)[31:0]);
    else
        bvr_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1()[31:0] == DBGBCR_EL1(n)[31:0]);
    end;
end;

```



```

end;

elsif match_cid1 then
    bvr_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() &&
        CONTEXTIDR_EL1() [31:0] == DBGEBR_EL1(n) [31:0]);
end;

if match_vmid then
    let vmid : bits(NUM_VMIDBITS) = VMID();
    var bvr_vmid : bits(NUM_VMIDBITS);

    if !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2().VS == '0' then
        bvr_vmid = ZeroExtend(NUM_VMIDBITS)(DBGEBR_EL1(n) [39:32]);
    else
        bvr_vmid = DBGEBR_EL1(n) [47:32];
    end;

    bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() && vmid == bvr_vmid);
end;

elsif match_cid2 then
    bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() &&
        DBGEBR_EL1(n) [63:32] == CONTEXTIDR_EL2() [31:0]);
end;

let bvr_match_valid : boolean = (match_addr || match_cid || match_cid1);
let bxvr_match_valid : boolean = (match_vmid || match_cid2);

var value_match : boolean = ((!bxvr_match_valid || bxvr_match) &&
    (!bvr_match_valid || bvr_match));

// A watchpoint might be linked to a linked address matching breakpoint with linking enabled,
// which is in turn linked to a context matching breakpoint.
if linked_to && linked then
    // If called from StateMatch and breakpoint is a linked breakpoint then it must be a
    // watchpoint that is linked to an address matching breakpoint which is linked to a
    // context matching breakpoint.
    assert !isbreakpnt && match_addr && IsFeatureImplemented(FEAT_ABLE);
    let lbnx : bits(2) = (if IsFeatureImplemented(FEAT_Debugv8p9) then DBGBCR_EL1(n).LBNX
        else '00');

    let linked_linked_n : integer{} = UInt(lbnx :: DBGBCR_EL1(n).LBN);
    var linked_value_match : boolean;
    let linked_vaddress = ARBITRARY : bits(64);
    let linked_linked_to = TRUE;
    let linked_isbreakpnt = TRUE;
    let linked_from_linking_enabled = TRUE;
    (bp_type, linked_value_match) = AArch64_BreakpointValueMatch(linked_linked_n,
        linked_vaddress,
        linked_linked_to,
        linked_isbreakpnt,
        linked_from_linking_enabled);

    value_match = value_match && linked_value_match;
end;

if match_addr && !mismatch then
    bp_type = BreakpointType_AdrMatch;
elsif match_addr && mismatch then
    bp_type = BreakpointType_AdrMismatch;
elsif match_vmid || match_cid || match_cid1 || match_cid2 then
    bp_type = BreakpointType_CtxtMatch;
else
    unreachable;
end;

return (bp_type, value_match);
end;

```

Library pseudocode for aarch64/debug/breakpoint/AArch64_ReservedBreakpointType

```
// AArch64_ReservedBreakpointType()
// =====
// Checks if the given DBGBCR_EL1(n).[BT2,BT] values are reserved and will
// generate Constrained Unpredictable behavior, otherwise returns Constraint_NONE.

func AArch64_ReservedBreakpointType(n : integer, bt_in : bits(5)) => (Constraint, bits(5))
begin
    var bt : bits(5)          = bt_in;
    var reserved : boolean    = FALSE;
    let context_aware : boolean = IsContextAwareBreakpoint(n);

    if bt[4] == '0' then
        // Context matching
        if bt != '00x0x' && !context_aware then
            reserved = TRUE;
        end;

        // EL2 extension
        if bt == '01xxx' && !HaveEL(EL2) then
            reserved = TRUE;
        end;

        // Context matching
        if (bt IN {'0011x', '011xx'}) && !IsFeatureImplemented(FEAT_VHE) &&
            !IsFeatureImplemented(FEAT_Debugv8p2) then
            reserved = TRUE;
        end;

        // Reserved
        if bt == '0010x' && !IsFeatureImplemented(FEAT_BWE) && !HaveAArch32EL(EL1) then
            reserved = TRUE;
        end;
    else
        // Reserved
        if bt != '10x0x' then
            reserved = TRUE;
        end;
    end;

    if reserved then
        var c : Constraint;
        (c, bt) = ConstrainUnpredictableBits{5}(Unpredictable RESBPTYPE);
        assert c IN {Constraint DISABLED, Constraint UNKNOWN};
        if c == Constraint DISABLED then
            return (c, ARBITRARY : bits(5));
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
        end;
    end;

    return (Constraint NONE, bt);
end;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64_StateMatch

```
// AArch64_StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

func AArch64_StateMatch(ssc_in : bits(2), ssce_in : bit, hmc_in : bit,
    pxc_in : bits(2), linked_in : boolean, linked_n_in : integer,
    isbreakpnt : boolean, vaddress : bits(64),
    accdesc : AccessDescriptor) => boolean

begin
    if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0'; end;

    // "ssc_in", "ssce_in", "hmc_in", "pxc_in" are the control fields from
    // the DBGBCR_EL1(n) or DBGWCR_EL1(n) register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR_EL1(n) or
    // DBGWCR_EL1(n) register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "vaddress" is the program counter for a linked watchpoint or the same value passed to
    // AArch64_CheckBreakpoint for a linked breakpoint.
    // "accdesc" describes the properties of the access being matched.
    var ssc : bits(2) = ssc_in;
    var ssce : bit = ssce_in;
    var hmc : bit = hmc_in;
    var pxc : bits(2) = pxc_in;
    let linked : boolean = linked_in;
    let linked_n : integer = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    var c : Constraint;
    (c, ssc, ssce, hmc, pxc) = CheckValidStateMatch(ssc, ssce, hmc, pxc, isbreakpnt);
    if c == Constraint DISABLED then return FALSE; end;
    // Otherwise the hmc, ssc, ssce, pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    let EL3_match : boolean = HaveEL(EL3) && hmc == '1' && ssc[0] == '0';
    let EL2_match : boolean = HaveEL(EL2) && ((hmc == '1' && (ssc::pxc != '1000')) || ssc == '11');
    let EL1_match : boolean = pxc[0] == '1';
    let EL0_match : boolean = pxc[1] == '1';

    var priv_match : boolean;
    case accdesc.el of
        when EL3 => priv_match = EL3_match;
        when EL2 => priv_match = EL2_match;
        when EL1 => priv_match = EL1_match;
        when EL0 => priv_match = EL0_match;
    end;

    // Security state match
    var ss_match : boolean;
    case ssce::ssc of
        when '000' => ss_match = hmc == '1' || accdesc.ss != SS Root;
        when '001' => ss_match = accdesc.ss == SS NonSecure;
        when '010' => ss_match = (hmc == '1' && accdesc.ss == SS Root) || accdesc.ss == SS Secure;
        when '011' => ss_match = (hmc == '1' && accdesc.ss != SS Root) || accdesc.ss == SS Secure;
        when '101' => ss_match = accdesc.ss == SS Realm;
    end;

    var linked_match : boolean = FALSE;

    if linked then
        // "linked_n" must be an enabled context-aware breakpoint unit. If it is not context-aware
        // then it is CONSTRAINED UNPREDICTABLE whether this gives no match, gives a match without
        // linking, or linked_n is mapped to some UNKNOWN breakpoint that is context-aware.
        if (!IsContextAwareBreakpoint(linked_n) &&
            (isbreakpnt || !IsAddressLinkingBreakpoint(linked_n))) then
            return ConstrainUnpredictableBool(Unpredictable BPNOTLINKING);
        end;
    end;

    if linked then
        let linked_to : boolean = TRUE;
        var bp_type : BreakpointType;
        let from_linking_enabled : boolean = FALSE;
        (bp_type, linked_match) = AArch64\_BreakpointValueMatch(linked_n, vaddress,
            linked_to, isbreakpnt,
            from_linking_enabled);

        if bp_type == BreakpointType AddrMismatch then
            linked_match = !linked_match;
        end;
    end;

    return priv_match && ss_match && (!linked || linked_match);
end;
```

Library pseudocode for aarch64/debug/breakpoint/DebugAddrTop

```
// DebugAddrTop()
// =====
// Returns the value for the top bit used in Breakpoint and Watchpoint address comparisons.

func DebugAddrTop() => AddressSize
begin
    if IsFeatureImplemented(FEAT_LVA3) then
        return 55;
    elsif IsFeatureImplemented(FEAT_LVA) then
        return 52;
    else
        return 48;
    end;
end;
```

Library pseudocode for aarch64/debug/breakpoint/EffectiveMDSELR_EL1_BANK

```
// EffectiveMDSELR_EL1_BANK()
// =====
// Return the effective value of MDSELR_EL1.BANK.

func EffectiveMDSELR_EL1_BANK() => bits(2)
begin
    // If 16 or fewer breakpoints and 16 or fewer watchpoints are implemented,
    // then the field is RES0.
    let num_bp : integer = NumBreakpointsImplemented();
    let num_wp : integer = NumWatchpointsImplemented();
    if num_bp <= 16 && num_wp <= 16 then
        return '00';
    end;

    // At EL3, the Effective value of this field is zero if MDCR_EL3.EBWE is 0.
    // At EL2, the Effective value is zero if the Effective value of MDCR_EL2.EBWE is 0.
    // That is, if either MDCR_EL3.EBWE is 0 or MDCR_EL2.EBWE is 0.
    // At EL1, the Effective value is zero if the Effective value of MDSCR_EL2.EMBWE is 0.
    // That is, if any of MDCR_EL3.EBWE, MDCR_EL2.EBWE, or MDSCR_EL1.EMBWE is 0.
    if ((HaveEL(EL3) && MDCR_EL3().EBWE == '0') ||
        (PSTATE.EL != EL3 && EL2Enabled() && MDCR_EL2().EBWE == '0') ||
        (PSTATE.EL == EL1 && MDSCR_EL1().EMBWE == '0')) then
        return '00';
    end;

    var bank : bits(2) = MDSELR_EL1().BANK;

    // Values are reserved depending on the number of breakpoints or watchpoints
    // implemented.
    if ((bank == '11' && num_bp <= 48 && num_wp <= 48) ||
        (bank == '10' && num_bp <= 32 && num_wp <= 32)) then
        // Reserved value
        (-, bank) = ConstrainUnpredictableBits{2}(Unpredictable_RESMDSELR);
        // The value returned by ConstrainUnpredictableBits must be a not-reserved value
    end;

    return bank;
end;
```

Library pseudocode for aarch64/debug/breakpoint/IsAddressLinkingBreakpoint

```
// IsAddressLinkingBreakpoint()
// =====
// Returns TRUE if DBGBCR_EL1(n) supports being configured as an Address matching breakpoint with
// linking enabled.

func IsAddressLinkingBreakpoint(n : integer) => boolean
begin
    return n < NumAddressLinkingBreakpointsImplemented();
end;
```

Library pseudocode for aarch64/debug/breakpoint/IsBreakpointEnabled

```
// IsBreakpointEnabled()
// =====
// Returns TRUE if the effective value of DBGBCR_EL1(n).E is '1', and FALSE otherwise.

func IsBreakpointEnabled(n : integer) => boolean
begin
    if (n > 15 &&
        ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEEnabled()) ||
         (HaltOnBreakpointOrWatchpoint() && EDSCR2().EHBWE == '0'))) then
        return FALSE;
    end;

    return DBGBCR_EL1(n).E == '1';
end;
```

Library pseudocode for aarch64/debug/breakpoint/NumAddressLinkingBreakpointsImplemented

```
// NumAddressLinkingBreakpointsImplemented()
// =====
// Returns the number of breakpoints that support being configured as Address matching breakpoint
// with linking enabled, as described by ID_AA64DFR1_EL1().ABL_CMPs.

func NumAddressLinkingBreakpointsImplemented() => integer
begin
    if !IsFeatureImplemented(FEAT_ABLE) then
        return 0;
    else
        return ImpDefInt("Number of address-linking breakpoints");
    end;
end;
```

Library pseudocode for aarch64/debug/breakpoint/SelfHostedExtendedBPWPEEnabled

```
// SelfHostedExtendedBPWPEEnabled()
// =====
// Returns TRUE if the extended breakpoints and watchpoints are enabled, and FALSE otherwise
// from a self-hosted debug perspective.

func SelfHostedExtendedBPWPEEnabled() => boolean
begin
    if NumBreakpointsImplemented() <= 16 && NumWatchpointsImplemented() <= 16 then
        return FALSE;
    end;

    if ((HaveEL(EL3) && MDCR_EL3().EBWE == '0') ||
        (EL2Enabled() && MDCR_EL2().EBWE == '0')) then
        return FALSE;
    end;

    return MDSCR_EL1().EMBWE == '1';
end;
```

Library pseudocode for aarch64/debug/ebep/CheckForPMUException

```
// CheckForPMUException()
// =====
// Take a PMU exception if enabled, permitted, and unmasked.

func CheckForPMUException()
begin
    var enabled : boolean;
    var target_el : bits(2);
    var pmu_exception : boolean;
    var synchronous : boolean;
    (enabled, target_el) = PMUExceptionEnabled();
    if !enabled || PMUExceptionMasked(target_el, PSTATE.EL, PSTATE.PM) then
        pmu_exception = FALSE;
    else
        let include_r1 : boolean = TRUE;
        let include_r2 : boolean = TRUE;
        let include_r3 : boolean = TRUE;
        pmu_exception = CheckPMUOverflowCondition(PMUOverflowCondition.PMUException,
                                                    include_r1, include_r2, include_r3);
        synchronous = FALSE;
    end;
    if pmu_exception then
        let fsc : bits(5) = '00000';
        TakeProfilingException(target_el, fsc, synchronous);
    end;
end;
```

Library pseudocode for aarch64/debug/ebep/EffectivePMEE

```
// EffectivePMEE()
// =====
// Return the Effective value of PMEE from MDCR_ELx or PMECR_EL1, handling reserved encodings:
// '10' (MDCR_ELx), '10' and '01' (PMECR_EL1).

func EffectivePMEE(el : bits(2), for_interrupt : boolean) => bits(2)
begin
    var val : bits(2);
    case el of
        when EL1 => val = PMECR_EL1().PMEE;
        when EL2 => val = MDCR_EL2().PMEE;
        when EL3 => val = MDCR_EL3().PMEE;
        otherwise => unreachable;
    end;

    if (val == '10' || (el == EL1 && val == '01')) then
        var c : Constraint;
        (c, val) = ConstrainUnpredictableBits{2}(Unpredictable RESPMEE);
        assert c IN {Constraint DISABLED, Constraint UNKNOWN};
        if c == Constraint DISABLED then
            val = if for_interrupt then '11' else '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        end;
    end;
    return val;
end;
```

Library pseudocode for aarch64/debug/ebep/PMUExceptionEnabled

```
// PMUExceptionEnabled()
// =====
// The first return value is TRUE if the PMU exception is enabled, and FALSE otherwise.
// The second return value is the target Exception level for an enabled PMU exception.

func PMUExceptionEnabled() => (boolean, bits(2))
begin

    if !IsFeatureImplemented(FEAT_EBEP) then
        return (FALSE, ARBITRARY : bits(2));
    end;

    var enabled : boolean;
    var target : bits(2) = ARBITRARY : bits(2);
    let for_interrupt : boolean = FALSE;
    let pmee_el3 : bits(2) = EffectivePMEE(EL3, for_interrupt);
    let pmee_el2 : bits(2) = EffectivePMEE(EL2, for_interrupt);
    let pmee_el1 : bits(2) = EffectivePMEE(EL1, for_interrupt);

    if HaveEL(EL3) && pmee_el3 != '01' then
        enabled = pmee_el3 == '11';
        if enabled then target = EL3; end;

    elsif EL2Enabled() && pmee_el2 != '01' then
        enabled = pmee_el2 == '11';
        if enabled then target = EL2; end;

    else
        enabled = pmee_el1 == '11';
        if enabled then
            target = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        end;
    end;

    return (enabled, target);
end;
```

Library pseudocode for aarch64/debug/ebep/PMUExceptionMasked

```
// PMUExceptionMasked()
// =====
// Return TRUE if the PMU Exception is masked at the specified target Exception level
// relative to the specified source Exception level, and by the value of PSTATE.PM,
// and FALSE otherwise.

func PMUExceptionMasked(target_el : bits(2), from_el : bits(2), pm : bit) => boolean
begin
    assert IsFeatureImplemented(FEAT_EBEP);
    let for_interrupt : boolean = FALSE;

    if Halted() then
        return TRUE;
    elsif UInt(target_el) < UInt(from_el) then
        return TRUE;
    elsif (from_el == EL2 && target_el == EL2 &&
           EffectivePMEE(EL2, for_interrupt) != '11') then
        return TRUE;
    elsif target_el == from_el && (PMECR_EL1().KPME == '0' || pm == '1') then
        return TRUE;
    end;

    return FALSE;
end;
```

Library pseudocode for aarch64/debug/ebep/PMUIInterruptEnabled

```
// PMUIInterruptEnabled()
// =====
// Return TRUE if the PMU interrupt request (PMUIRQ) is enabled, FALSE otherwise.

func PMUIInterruptEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_EBEP) then
        return TRUE;
    end;

    var enabled : boolean;
    let for_interrupt : boolean = TRUE;
    let pmee_el3 : bits(2) = EffectivePMEE(EL3, for_interrupt);
    let pmee_el2 : bits(2) = EffectivePMEE(EL2, for_interrupt);
    let pmee_el1 : bits(2) = EffectivePMEE(EL1, for_interrupt);

    if HaveEL(EL3) && pmee_el3 != '01' then
        enabled = pmee_el3 == '00';
    elsif EL2Enabled() && pmee_el2 != '01' then
        enabled = pmee_el2 == '00';
    else
        enabled = pmee_el1 == '00';
    end;

    return enabled;
end;
```

Library pseudocode for aarch64/debug/enables/AArch64_GenerateDebugExceptions

```
// AArch64_GenerateDebugExceptions()
// =====

func AArch64_GenerateDebugExceptions() => boolean
begin
    let ss : SecurityState = CurrentSecurityState();
    return AArch64_GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D);
end;
```

Library pseudocode for aarch64/debug/enables/AArch64_GenerateDebugExceptionsFrom

```
// AArch64_GenerateDebugExceptionsFrom()
// =====

func AArch64_GenerateDebugExceptionsFrom(from_el : bits(2), from_state : SecurityState,
                                         mask : bit) => boolean
begin
    if OSLSR_EL1().OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    end;

    let route_to_el2 : boolean = (HaveEL(EL2) &&
                                (from_state != SS Secure || IsSecureEL2Enabled()) &&
                                (HCR_EL2().TGE == '1' || MDCR_EL2().TDE == '1'));
    let target : bits(2) = (if route_to_el2 then EL2 else EL1);
    var enabled : boolean;
    if HaveEL(EL3) && from_state == SS Secure then
        enabled = MDCR_EL3().SDD == '0';
        if from_el == ELO && ELUsingAArch32(EL1) then
            enabled = enabled || SDER32_EL3().SUIDEN == '1';
        end;
    else
        enabled = TRUE;
    end;

    if from_el == target then
        enabled = enabled && MDSCR_EL1().KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);
    end;

    return enabled;
end;
```

Library pseudocode for aarch64/debug/ite/AArch64_TRCIT

```
// AArch64_TRCIT()
// =====
// Determines whether an Instrumentation trace packet should
// be generated and then generates an instrumentation trace packet
// containing the value of the register passed as an argument

func AArch64_TRCIT(Xt : bits(64))
begin
    let ss = CurrentSecurityState();
    if TraceInstrumentationAllowed(ss, PSTATE.EL) then
        TraceInstrumentation(Xt);
    end;
end;
```

Library pseudocode for aarch64/debug/ite/TraceInstrumentation

```
// TraceInstrumentation()
// =====
// Generates an instrumentation trace packet
// containing the value of the register passed as an argument

impdef func TraceInstrumentation(Xt : bits(64))
begin
    return;
end;
```


Library pseudocode for aarch64/debug/pmu/AArch64_IncrementCycleCounter

```
// AArch64_IncrementCycleCounter()
// =====
// Increment the cycle counter and possibly set overflow bits.

func AArch64_IncrementCycleCounter()
begin
    if !CountPMUEvents(CYCLE_COUNTER_ID) then return; end;
    var d : bit = PMCR_EL0().D; // Check divide-by-64
    var lc : bit = PMCR_EL0().LC;
    var lc_enabled : boolean;
    (lc_enabled, -) = PMUExceptionEnabled();
    lc = if lc_enabled then '1' else lc;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0'; end;
    if d == '1' && !HasElapsed64Cycles() then return; end;

    let old_value : integer = UInt(PMCCNTR_EL0());
    let new_value : integer = old_value + 1;
    PMCCNTR_EL0() = new_value[63:0];

    let ovflw : integer{} = if HaveAArch32() && lc == '0' then 32 else 64;

    if old_value[64:ovflw] != new_value[64:ovflw] then
        PMOVSSET_EL0().C = '1';
    end;

    return;
end;
```

Library pseudocode for aarch64/debug/pmu/AArch64_IncrementEventCounter

```
// AArch64_IncrementEventCounter()
// =====
// Increment the specified event counter 'idx' by the specified amount 'increment'.
// 'Vm' is the value event counter 'idx-1' is being incremented by if 'idx' is odd,
// zero otherwise.
// Returns the amount the counter was incremented by.

func AArch64_IncrementEventCounter(idx : integer, increment_in : integer, Vm : integer) => integer
begin
    var old_value : integer;
    var new_value : integer;

    old_value = UInt(PMEVCNTR_EL0(idx));
    let increment : integer = PMUCountValue(idx, increment_in, Vm);
    new_value = old_value + increment;

    var lp : bit;
    if IsFeatureImplemented(FEAT_PMuV3p5) then
        PMEVCNTR_EL0(idx) = new_value[63:0];
        var pmuexception_enabled : boolean;
        (pmuexception_enabled, -) = PMUExceptionEnabled();
        if pmuexception_enabled then
            lp = '1';
        else
            case GetPMUCounterRange(idx) of
                when PMUCounterRange_R1 =>
                    lp = PMCR_EL0().LP;
                when PMUCounterRange_R2 =>
                    lp = MDCR_EL2().HLP;
                when PMUCounterRange_R3 =>
                    lp = '1';
                otherwise =>
                    unreachable;
            end;
        end;
    else
        lp = '0';
        PMEVCNTR_EL0(idx) = ZeroExtend{64}(new_value[31:0]);
    end;

    let ovflw : integer{} = if lp == '1' then 64 else 32;
    if old_value[64:ovflw] != new_value[64:ovflw] then
        PMOVSSET_EL0()[idx] = '1';
        // Check for the CHAIN event from an even counter
        if (idx[0] == '0' && idx + 1 < NUM_PMU_COUNTERS && lp == '0' &&
            (GetPMUCounterRange(idx) == GetPMUCounterRange(idx+1) ||
             ConstrainUnpredictableBool(Unpredictable_COUNT_CHAIN))) then
            // If PMU counters idx and idx+1 are not in same range,
            // it is CONSTRAINED UNPREDICTABLE if CHAIN event is counted
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
        end;
    end;

    return increment;
end;
```

Library pseudocode for aarch64/debug/pmu/AArch64_PMUCycle

```
// AArch64_PMUCycle()
// =====
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

func AArch64_PMUCycle()
begin
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    end;
    PMUEvent(PMU_EVENT_CPU_CYCLES);
    let counters : integer = NUM_PMU_COUNTERS;
    var Vm : integer = 0;
    if counters != 0 then
        for idx = 0 to counters - 1 do
            if CountPMUEvents(idx) then
                let accumulated : integer = PMUEventAccumulator[[idx]];
                if (idx MOD 2) == 0 then Vm = 0; end;
                Vm = AArch64_IncrementEventCounter(idx, accumulated, Vm);
            end;
            PMUEventAccumulator[[idx]] = 0;
        end;
    end;
    AArch64_IncrementCycleCounter();
    CheckForPMUOverflow();
end;
```

Library pseudocode for aarch64/debug/profilingexception/TakeProfilingException

```
// TakeProfilingException()
// =====
// Takes a Profiling exception.

func TakeProfilingException(target_el : bits(2), fsc : bits(5), synchronous : boolean)
begin
    var except : ExceptionRecord = ExceptionSyndrome(Exception_Profiling);
    except.syndrome.iss[5:1] = fsc;
    if synchronous then
        except.syndrome.iss[0] = '1';
    end;

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CheckForSPEException

```
// CheckForSPEException()
// =====
// Take an SPE Profiling exception if pending, permitted, and unmasked.

func CheckForSPEException()
begin
    if !IsFeatureImplemented(FEAT_SPE_EXC) then
        return;
    end;

    if Halted\(\) || Restarting\(\) then
        return;
    end;

    var route_to_el3 : boolean = FALSE;
    var route_to_el2 : boolean = FALSE;
    var route_to_el1 : boolean = FALSE;

    if HaveEL\(EL3\) && MDCR_EL3().PMSEE == '1x' then
        let pending : boolean = PMBSR_EL3().S == '1';
        let masked : boolean = PSTATE.EL == EL3;
        route_to_el3 = pending && !masked;
    end;

    var owning_ss : SecurityState;
    var owning_el : bits(2);
    (owning_ss, owning_el) = ProfilingBufferOwner\(\);
    let in_owning_ss : boolean = IsCurrentSecurityState(owning_ss);

    if EffectivePMSCR\_EL2\_EE\(\) == '1x' then
        let pending : boolean = PMBSR_EL2().S == '1';
        let masked : boolean = (!in_owning_ss || PSTATE.EL == EL3 ||
            (PSTATE.EL == EL2 && (PMSCR_EL2().EE != '11' ||
                PMSCR_EL2().KE == '0' || PSTATE.PM == '1')))
        );
        route_to_el2 = pending && !masked;
    end;

    if EffectivePMSCR\_EL1\_EE\(\) == '11' then
        let pending : boolean = PMBSR_EL1().S == '1';
        let masked : boolean = (!in_owning_ss || PSTATE.EL IN {EL3, EL2} ||
            (PSTATE.EL == EL1 && (PMSCR_EL1().KE == '0' || PSTATE.PM == '1')))
        );
        if EffectiveTGE\(\) == '1' then
            route_to_el2 = route_to_el2 || (pending && !masked);
        else
            route_to_el1 = pending && !masked;
        end;
    end;

    let fsc : bits(5) = '00001'; // SPE exception
    let synchronous : boolean = FALSE;

    // The relative priorities of the following checks is IMPLEMENTATION DEFINED
    if route_to_el3 then
        TakeProfilingException\(EL3, fsc, synchronous\);
    end;
    if route_to_el2 then
        TakeProfilingException\(EL2, fsc, synchronous\);
    end;
    if route_to_el1 then
        TakeProfilingException\(EL1, fsc, synchronous\);
    end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CheckMDCR_EL3_NSPBTrap

```
// CheckMDCR_EL3_NSPBTrap()
// =====
// Check if the register access is trappable by MDCR_EL3.[NSPBE, NSPB]

func CheckMDCR_EL3_NSPBTrap() => boolean
begin
    var state_bits : bits(3);
    var reserved : boolean;
    (state_bits, reserved) = EffectiveMDCR\_EL3\_NSPB\(\);
    return ((reserved && ConstrainUnpredictableBool(Unpredictable\_RESERVEDNSxB\_Trap)) ||
        state_bits[0] == '0' || state_bits[1] != EffectiveSCR\_EL3\_NS\(\) ||
        (IsFeatureImplemented(FEAT_RME) && state_bits[2] != EffectiveSCR\_EL3\_NSE\(\)));
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// =====
func CollectContextIDR1() => boolean
begin
    if !StatisticalProfilingEnabled() then return FALSE; end;
    if PSTATE.EL == EL2 then return FALSE; end;
    if EL2Enabled() && HCR_EL2().TGE == '1' then return FALSE; end;
    return PMSCR_EL1().CX == '1';
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// =====
func CollectContextIDR2() => boolean
begin
    if !StatisticalProfilingEnabled() then return FALSE; end;
    if !EL2Enabled() then return FALSE; end;
    return PMSCR_EL2().CX == '1';
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// =====
func CollectPhysicalAddress() => boolean
begin
    if !StatisticalProfilingEnabled() then return FALSE; end;
    let (owning_ss, owning_el) : (SecurityState, bits(2)) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        return PMSCR_EL2().PA == '1' && (owning_el == EL2 || PMSCR_EL1().PA == '1');
    else
        return PMSCR_EL1().PA == '1';
    end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// =====
func CollectTimeStamp() => TimeStamp
begin
    if !StatisticalProfilingEnabled() then return TimeStamp None; end;
    let (-, owning_el) = ProfilingBufferOwner();

    if owning_el == EL2 then
        if PMSCR_EL2().TS == '0' then return TimeStamp None; end;
    else
        if PMSCR_EL1().TS == '0' then return TimeStamp None; end;
    end;

    var PCT_el1 : bits(2);
    if !IsFeatureImplemented(FEAT_ECV) then
        PCT_el1 = '0'::PMSCR_EL1().PCT[0]; // PCT[1] is RES0
    else
        PCT_el1 = PMSCR_EL1().PCT;
        if PCT_el1 == '10' then
            // Reserved value
            (-, PCT_el1) = ConstrainUnpredictableBits{2}(Unpredictable PMSCR_PCT);
        end;
    end;

    if EL2Enabled() then
        var PCT_el2 : bits(2);
        if !IsFeatureImplemented(FEAT_ECV) then
            PCT_el2 = '0'::PMSCR_EL2().PCT[0]; // PCT[1] is RES0
        else
            PCT_el2 = PMSCR_EL2().PCT;
            if PCT_el2 == '10' then
                // Reserved value
                (-, PCT_el2) = ConstrainUnpredictableBits{2}(Unpredictable PMSCR_PCT);
            end;
        end;

        case PCT_el2 of
            when '00' =>
                return if IsInHost() then TimeStamp Physical else TimeStamp Virtual;
            when '01' =>
                if owning_el == EL2 then return TimeStamp Physical; end;
            when '11' =>
                assert IsFeatureImplemented(FEAT_ECV); // FEAT_ECV must be implemented
                if owning_el == EL1 && PCT_el1 == '00' then
                    return if IsInHost() then TimeStamp Physical else TimeStamp Virtual;
                else
                    return TimeStamp OffsetPhysical;
                end;
            otherwise =>
                unreachable;
        end;
    end;

    case PCT_el1 of
        when '00' => return if IsInHost() then TimeStamp Physical else TimeStamp Virtual;
        when '01' => return TimeStamp Physical;
        when '11' =>
            assert IsFeatureImplemented(FEAT_ECV); // FEAT_ECV must be implemented
            return TimeStamp OffsetPhysical;
        otherwise => unreachable;
    end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/DefaultSPEEvent

```
// DefaultSPEEvent()
// =====
// Return the target ELx for an indirect write to PMBSR_ELx for an Other buffer management
// event or anything other than a buffer management event.

func DefaultSPEEvent() => bits(2)
begin
    return ReportSPEEvent(Zeros{6}, ARBITRARY : bits(6));
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/EffectiveMDCR_EL3_NSPB

```
// EffectiveMDCR_EL3_NSPB()
// =====
// Return the Effective value of MDCR_EL3.[NSPBE, NSPB] field and whether it is a reserved value.

readonly func EffectiveMDCR_EL3_NSPB() => (bits(3), boolean)
begin
    var state_bits : bits(3);
    var reserved : boolean = FALSE;

    if IsFeatureImplemented(FEAT_RME) then
        state_bits = MDCR_EL3().[NSPBE, NSPB];
        if state_bits == '10x' || (!IsFeatureImplemented(FEAT_Secure) && state_bits == '00x') then
            // Reserved value
            reserved = TRUE;
            (-, state_bits) = ConstrainUnpredictableBits{3}(Unpredictable\_RESERVEDNSxB);
        end;
    else
        state_bits = '0' :: MDCR_EL3().NSPB;
    end;

    return (state_bits, reserved);
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/EffectivePMBLIMITR_EL1_nVM

```
// EffectivePMBLIMITR_EL1_nVM()
// =====

func EffectivePMBLIMITR_EL1_nVM() => bit
begin
    if !IsFeatureImplemented(FEAT_SPE_nVM) then
        return '0';
    end;
    if HaveEL\(EL2\) then
        let (owning_ss, owning_el) : (SecurityState, bits(2)) = ProfilingBufferOwner();
        if ((owning_ss != SS\_Secure || IsSecureEL2Enabled()) && owning_el == EL1 &&
            PMSCR_EL2().EnVM == '0') then
            return '0';
        end;
    end;
    return PMBLIMITR_EL1().nVM;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/EffectivePMSCR_EL1_EE

```
// EffectivePMSCR_EL1_EE()
// =====
// Return the Effective value of PMSCR_EL1.EE for the purpose of controlling the
// SPE Profiling exception.

func EffectivePMSCR_EL1_EE() => bits(2)
begin
    if EffectivePMSCR\_EL2\_EE() == '00' then
        return '00';
    end;

    var ee : bits(2) = PMSCR_EL1().EE;
    if ee IN {'01', '10'} then // Reserved value
        if IsFeatureImplemented(FEAT_NV) then
            ee[0] = ee[1];
        else
            var c : Constraint;
            (c, ee) = ConstrainUnpredictableBits{2}(Unpredictable\_RESPMSEE);
            assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
            if c == Constraint\_DISABLED then
                ee = '00';
            end;
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        end;
    end;

    return ee;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/EffectivePMSCR_EL2_EE

```
// EffectivePMSCR_EL2_EE()
// =====
// Return the Effective value of PMSCR_EL2.EE.

readonly func EffectivePMSCR_EL2_EE() => bits(2)
begin
    if !IsFeatureImplemented(FEAT_SPE_EXC) then
        return '00';
    end;

    if HaveEL(EL3) && MDCR_EL3().PMSEE == '00' then
        return '00';
    end;

    let check_el2 : boolean = (HaveEL(EL2) &&
        (EffectiveSCR_EL3_NS() == '1' || IsSecureEL2Enabled()));
    return if check_el2 then PMSCR_EL2().EE else '01';
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/GetPMBSR_EL1_FSC

```
// GetPMBSR_EL1_FSC()
// =====
// Query the PMBSR_EL1.FSC field.

func GetPMBSR_EL1_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = PMBSR_EL1()[5:0];
    return FSC;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/GetPMBSR_EL2_FSC

```
// GetPMBSR_EL2_FSC()
// =====
// Query the PMBSR_EL2.FSC field.

func GetPMBSR_EL2_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = PMBSR_EL2()[5:0];
    return FSC;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/GetPMBSR_EL3_FSC

```
// GetPMBSR_EL3_FSC()
// =====
// Query the PMBSR_EL3.FSC field.

func GetPMBSR_EL3_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = PMBSR_EL3()[5:0];
    return FSC;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/OtherSPERManagementEvent

```
// OtherSPERManagementEvent()
// =====
// Report an Other buffer management event, with the status code 'bsc'

func OtherSPERManagementEvent(bsc : bits(6))
begin
    let target_el : bits(2) = DefaultSPEEvent();
    if PMBSR_EL(target_el).S == '0' then
        PMBSR_EL(target_el).S = '1'; // Assert interrupt or exception
        PMBSR_EL(target_el).EC = '000000'; // Other buffer management event
        PMBSR_EL(target_el).MSS = ZeroExtend{16}(bsc);
        PMBSR_EL(target_el).MSS2 = Zeros{24};
    end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/PMBSR_EL

```
// PMBSR_EL - accessor
// =====

accessor PMBSR_EL(el : bits(2)) <=> value : PMBSRType
begin
  getter
    var r : bits(64);
    case el of
      when EL1 => r = PMBSR_EL1();
      when EL2 => r = PMBSR_EL2();
      when EL3 => r = PMBSR_EL3();
      otherwise => unreachable;
    end;
    return r;
  end;
  setter
    let r : bits(64) = value;
    case el of
      when EL1 => PMBSR_EL1() = r;
      when EL2 => PMBSR_EL2() = r;
      when EL3 => PMBSR_EL3() = r;
      otherwise => unreachable;
    end;
    return;
  end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// =====

readonly func ProfilingBufferEnabled() => boolean
begin
  if !IsFeatureImplemented(FEAT_SPE) then
    return FALSE;
  end;

  let (-, owning_el) : (SecurityState, bits(2)) = ProfilingBufferOwner();

  return !ELUsingAArch32(owning_el) && PMBLIMITR_EL1().E == '1';
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// =====

readonly func ProfilingBufferOwner() => (SecurityState, bits(2))
begin
  var owning_ss : SecurityState;

  var state_bits : bits(3);
  if HaveEL(EL3) then
    (state_bits, -) = EffectiveMDCR\_EL3\_NSPB();
  else
    state_bits = if SecureOnlyImplementation() then '001' else '011';
  end;

  case state_bits of
    when '00x' => owning_ss = SS\_Secure;
    when '01x' => owning_ss = SS\_NonSecure;
    when '11x' => owning_ss = SS\_Realm;
  end;

  var owning_el : bits(2);
  if HaveEL(EL2) && (owning_ss != SS\_Secure || IsSecureEL2Enabled()) then
    owning_el = if MDCR_EL2().E2PB == '00' then EL2 else EL1;
  else
    owning_el = EL1;
  end;

  return (owning_ss, owning_el);
end;
```


Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// ProfilingSynchronizationBarrier()
// =====
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.

impdef func ProfilingSynchronizationBarrier()
begin
    return;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/ReportSPEEvent

```
// ReportSPEEvent()
// =====
// Return the target ELx for an indirect write to PMBSR_ELx.
// When the indirect write is due to a buffer management event:
// 'ec_bits' is the Event Class for the management event.
// 'fsc_bits' is the Fault Status Code when this is a fault, ignored otherwise.
// Otherwise, 'ec_bits' should be Zeros().

func ReportSPEEvent(ec_bits : bits(6), fsc_bits : bits(6)) => bits(2)
begin
    var target_el : bits(2);
    var route_to_el3 : boolean = FALSE;
    var route_to_el2 : boolean = FALSE;

    if IsFeatureImplemented(FEAT_SPE_EXC) then
        if ec_bits == '011111' then // implementation defined fault
            return ImpDefBits{2}("target_el for SPE ImpDef Fault");
        end;

        let s1fault : boolean = (ec_bits == '100100'); // Stage 1 fault
        let s2fault : boolean = (ec_bits == '100101'); // Stage 2 fault

        var gpcfault, gpfault : boolean;
        if IsFeatureImplemented(FEAT_RME) then
            // Granule Protection Check fault, other than GPF. That is, a GPT address size fault,
            // GPT walk fault, or synchronous External abort on GPT fetch.
            gpcfault = (ec_bits == '011110');
            // Other Granule Protection Fault, reported as Stage 1 or Stage 2 fault.
            gpfault = ((s1fault || s2fault) && fsc_bits IN {'10001x', '1001xx', '101000'});
        else
            gpcfault = FALSE;
            gpfault = FALSE;
        end;
        let sync_ext_abort : boolean = ((s1fault || s2fault) &&
            fsc_bits IN {'010000', '01001x', '0101xx', '011011'});

        var owning_ss : SecurityState;
        var owning_el : bits(2);
        (owning_ss, owning_el) = ProfilingBufferOwner();

        if HaveEL\(EL3\) && MDCR_EL3().PMSEE == '1x' then
            route_to_el3 = (MDCR_EL3().PMSEE == '11' ||
                gpcfault || (gpfault && SCR_EL3().GPF == '1') ||
                (sync_ext_abort && EffectiveEA() == '1'));
        end;

        if EffectivePMSR\_EL2\_EE() == '1x' then
            route_to_el2 = (PMSR_EL2().EE == '11' || (s1fault && owning_el == EL2) || s2fault ||
                gpcfault || (gpfault && HCR_EL2().GPF == '1') ||
                (sync_ext_abort && EffectiveHCR\_TEA() == '1'));
        end;
    end;

    if route_to_el3 then
        target_el = EL3;
    elseif route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;
    end;

    return target_el;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPE

```
// SPE Implementation Constants
// =====

constant SPEMaxAddrs : integer{} = 32;
constant SPEMaxCounters : integer{} = 32;
constant SPEMaxRecordSize : integer{} = 2048;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEAddByteToRecord

```
// SPEAddByteToRecord()
// =====
// Add one byte to a record and increase size property appropriately.

func SPEAddByteToRecord(b : bits(8))
begin
    assert SPERecordSize < SPEMaxRecordSize;
    SPERecordData[[SPERecordSize]] = b;
    SPERecordSize = SPERecordSize + 1;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEAddPacketToRecord

```
// SPEAddPacketToRecord()
// =====
// Add passed header and payload data to the record.
// Payload must be a multiple of 8.

func SPEAddPacketToRecord{N}(header_hi : bits(2), header_lo : bits(4),
                             payload : bits(N))
begin
    assert N MOD 8 == 0;
    var sz : bits(2);
    case N of
        when 8 => sz = '00';
        when 16 => sz = '01';
        when 32 => sz = '10';
        when 64 => sz = '11';
        otherwise => unreachable;
    end;

    let header : bits(8) = header_hi::sz::header_lo;
    SPEAddByteToRecord(header);
    for i = 0 to (N DIV 8)-1 do
        SPEAddByteToRecord(payload[i*8]);
    end;
end;
```



```

// SPEBranch()
// =====
// Called on every branch if SPE is present. Maintains previous branch target
// and branch related SPE functionality.

func SPEBranch{N}(target : bits(N), branch_type : BranchType, conditional : boolean,
    taken_flag : boolean)
begin
    let is_isb : boolean = FALSE;
    SPEBranch{N}(target, branch_type, conditional, taken_flag, is_isb);
end;

func SPEBranch{N}(target : bits(N), branch_type : BranchType, conditional : boolean,
    taken_flag : boolean, is_isb : boolean)
begin
    // If the PE implements branch prediction, data about (mis)prediction is collected
    // through the PMU events.

    var collect_prev_br : boolean;
    let collect_prev_br_eret : boolean = ImpDefBool("SPE prev br oneret");
    let collect_prev_br_exception : boolean = (ImpDefBool(
        "SPE prev br on exception"));
    let collect_prev_br_isb : boolean = ImpDefBool("SPE prev br on isb");
    case branch_type of
        when BranchType EXCEPTION => collect_prev_br = collect_prev_br_exception;
        when BranchType ERET => collect_prev_br = collect_prev_br_eret;
        otherwise => collect_prev_br = !is_isb || collect_prev_br_isb;
    end;

    // Implements previous branch target functionality
    if (taken_flag && IsFeatureImplemented(FEAT_SPE_PBT) && StatisticalProfilingEnabled() &&
        collect_prev_br) then

        if SPESampleInFlight then
            let previous_target : bits(64) = SPESamplePreviousBranchAddress;
            SPESampleAddress[[SPEAddrPosPrevBranchTarget]][63:0] = previous_target[63:0];
            let previous_branch_valid : boolean = SPESamplePreviousBranchAddressValid;
            SPESampleAddressValid[[SPEAddrPosPrevBranchTarget]] = previous_branch_valid;
        end;
        // Save the target address for it to be added to a future record.
        SPESamplePreviousBranchAddress[55:0] = target[55:0];

        var ns : bit;
        var nse : bit;
        case CurrentSecurityState() of
            when SS Secure =>
                ns = '0';
                nse = '0';
            when SS NonSecure =>
                ns = '1';
                nse = '0';
            when SS Realm =>
                ns = '1';
                nse = '1';
            otherwise => unreachable;
        end;

        SPESamplePreviousBranchAddress[63] = ns;
        SPESamplePreviousBranchAddress[60] = nse;
        SPESamplePreviousBranchAddress[62:61] = PSTATE.EL;
        SPESamplePreviousBranchAddressValid = TRUE;
    end;

    if !StatisticalProfilingEnabled() then
        if taken_flag then
            // Invalidate previous branch address, if profiling is disabled
            // or prohibited.
            SPESamplePreviousBranchAddressValid = FALSE;
        end;
        return;
    end;

    if SPESampleInFlight then
        SPESampleOpAttr.branch_is_direct = branch_type IN {BranchType DIR, BranchType DIRCALL};
        SPESampleOpAttr.branch_has_link = branch_type IN {BranchType DIRCALL, BranchType INDCALL};
        SPESampleOpAttr.procedure_return = branch_type == BranchType RET;
        SPESampleOpAttr.op_type = SPEOpType Branch;
        SPESampleOpAttr.is_conditional = conditional;
        SPESampleOpAttr.cond_pass = taken_flag;

        // Save the target address.
        if taken_flag then
            var ns : bit;
            var nse : bit;
            case CurrentSecurityState() of
                when SS Secure =>
                    ns = '0';
                    nse = '0';
                when SS NonSecure =>

```

```

        ns = '1';
        nse = '0';
    when SS\_Realm =>
        ns = '1';
        nse = '1';
    otherwise => unreachable;
end;
let el : bits(2) = PSTATE.EL;

SPESampleAddress[[SPEAddrPosBranchTarget]][55:0] = target[55:0];
SPESampleAddress[[SPEAddrPosBranchTarget]][63:56] = ns::el::nse::Zeros{4};
SPESampleAddressValid[[SPEAddrPosBranchTarget]] = TRUE;
end;
end;
end;

```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEBufferIsFull

```

// SPEBufferIsFull()
// =====
// Return true if another full size sample record would not fit in the
// profiling buffer.

func SPEBufferIsFull() => boolean
begin
    let write_pointer_limit : integer = UInt(PMBLIMITR_EL1().LIMIT::Zeros{12});
    let current_write_pointer : integer = UInt(PMBPTR_EL1());
    let record_max_size : integer = 1 << UInt(PMSIDR_EL1().MaxSize);
    return current_write_pointer > (write_pointer_limit - record_max_size);
end;

```

Library pseudocode for aarch64/debug/statisticalprofiling/SPECollectRecord

```

// SPECollectRecord()
// =====
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

func SPECollectRecord(events : bits(64), total_latency : integer, optype : SPEOpType) => boolean
begin
    // Filtering by events
    var is_rejected_event : boolean;
    var is_rejected_nevent : boolean;
    (is_rejected_event, is_rejected_nevent) = SPEFilterByEvents(events);

    // Filtering by type
    let is_rejected_type : boolean = SPEFilterByType(SPESampleOpAttr);

    // Filtering by latency
    let is_rejected_latency : boolean = SPEFilterByLatency(total_latency);

    let is_rejected_data_source : boolean = SPEFilterByDataSource(optype);

    var return_value : boolean;
    return_value = !(is_rejected_nevent || is_rejected_event ||
                    is_rejected_type || is_rejected_latency ||
                    is_rejected_data_source);

    if return_value then
        PMUEvent(PMU_EVENT_SAMPLE_FILTRATE);
    end;

    return return_value;
end;

```



```

// SPECompleteSample()
// =====
// Called to complete the sampling process.

func SPECompleteSample()
begin
    assert SPESampleInFlight;

    PMUEvent (PMU_EVENT_SAMPLE_FEED);

    // Stop any pending counters
    for counter_index = 0 to (SPEMaxCounters - 1) do
        if SPESampleCounterPending[[counter_index]] then
            SPEStopCounter (counter_index);
        end;
    end;

    // Record any IMPLEMENTATION DEFINED events
    let impdef_events : bits(64) = ImpDefBits{}("SPE EVENTS");
    SPESampleEvents[63:48] = impdef_events[63:48];
    // The number of bits available for IMPLEMENTATION DEFINED events
    // is reduced by FEAT_SPEv1p4
    if !IsFeatureImplemented(FEAT_SPEv1p4) then
        SPESampleEvents[31:24] = impdef_events[31:24];
    end;
    SPESampleEvents[15:12] = impdef_events[15:12];
    // Bit 24 encodes whether the sample was collected in Streaming SVE mode.
    if IsFeatureImplemented(FEAT_SPE_SME) then
        SPESampleEvents[24] = PSTATE.SM;
    end;
    SPESampleEvents[6] = if (SPESampleOpAttr.is_conditional &&
        !SPESampleOpAttr.cond_pass) then '1' else '0';

    var collect_record : boolean;
    collect_record = SPECollectRecord(SPESampleEvents,
        SPESampleCounter[[SPECounterPosTotalLatency]],
        SPESampleOpAttr.op_type);

    var discard : boolean = FALSE;
    if IsFeatureImplemented(FEAT_SPEv1p2) then
        discard = PMBLIMITR\_EL1() .FM == '10';
    end;
    if collect_record && !discard then
        SPEConstructRecord();
        if SPEBufferIsFull() then
            let bsc : bits(6) = '000001'; // Buffer full event
            OtherSPEManagementEvent (bsc);
            PMUEvent (PMU_EVENT_SAMPLE_BUFFER_FULL);
        end;
    end;

    SPESampleInFlight = FALSE;

    SPEResetSampleStorage();
end;

// Order of addresses in sample storage and architectural index values
constant SPEAddrPosPCVirtual : integer = 0;
constant SPEAddrPosBranchTarget : integer = 1;
constant SPEAddrPosDataVirtual : integer = 2;
constant SPEAddrPosDataPhysical : integer = 3;
constant SPEAddrPosPrevBranchTarget : integer = 4;
// Order of counters in sample storage and architectural index values
constant SPECounterPosTotalLatency : integer = 0;
constant SPECounterPosIssueLatency : integer = 1;
constant SPECounterPosTranslationLatency : integer = 2;
constant SPECounterPosAltIssueLatency : integer = 4;
// Sample in-flight flag
var SPESampleInFlight : boolean = FALSE;

// Globally declared variables which store data about the sample.

// Context storage
var SPESampleContextEL1 : bits(32);
var SPESampleContextEL1Valid : boolean;
var SPESampleContextEL2 : bits(32);
var SPESampleContextEL2Valid : boolean;

// Counter storage
var SPESampleCounter : array [[SPEMaxCounters]] of integer;
var SPESampleCounterValid : array [[SPEMaxCounters]] of boolean;
var SPESampleCounterPending : array [[SPEMaxCounters]] of boolean;

// Address storage
var SPESampleAddress : array [[SPEMaxAddrs]] of bits(64);
var SPESampleAddressValid : array [[SPEMaxAddrs]] of boolean;
var SPESamplePreviousBranchAddress : bits(64);
var SPESamplePreviousBranchAddressValid : boolean;

// Data source storage

```

```
var SPESampleDataSource : bits(16);
var SPESampleDataSourceValid : boolean;

// OpAttr storage
var SPESampleOpAttr : SPEOpAttr;

// Timestamp storage
var SPESampleTimestamp : bits(64);
var SPESampleTimestampValid : boolean;

// Event storage
var SPESampleEvents : bits(64);
```


Library pseudocode for aarch64/debug/statisticalprofiling/SPEConstructClass

```
// SPEConstructClass()
// =====
// Constructs the encodings for the class and subclass fields.

func SPEConstructClass() => (bits(2), bits(8))
begin
  var op_class : bits(2);
  var op_subclass : bits(8);
  let cond : bit = if SPESampleOpAttr.is\_conditional then '1' else '0';
  let fp : bit = if SPESampleOpAttr.is\_floating\_point then '1' else '0';
  let ldst : bit = if SPESampleOpAttr.op\_type == SPEOpType\_Store then '1' else '0';
  let ar : bit = if SPESampleOpAttr.is\_acquire\_release then '1' else '0';
  let excl : bit = if SPESampleOpAttr.is\_exclusive then '1' else '0';
  let at : bit = if SPESampleOpAttr.at then '1' else '0';
  let indirect : bit = if SPESampleOpAttr.branch is direct then '0' else '1';
  let pred : bit = if SPESampleOpAttr.is\_predicated then '1' else '0';
  let sg : bit = if SPESampleOpAttr.is\_gather\_scatter then '1' else '0';
  let evl : bits(3) = SPESampleOpAttr.evl;
  let simd : bit = if SPESampleOpAttr.is\_simd then '1' else '0';
  let ets : bits(4) = SPESampleOpAttr.ets;
  // Since this implementation of SPE samples instruction instead of micro-operations, a
  // Branch with link or Procedure return instruction will never be recorded as a "Load/store,
  // GCS" format Operation Type packet. Therefore the COMMON bit is hard-wired to '1'.
  let common : bit = '1';
  if SPESampleOpAttr.op\_type == SPEOpType\_Other then
    op_class = '00';
    op_subclass = Zeros{5}::simd::fp::cond;
  elseif SPESampleOpAttr.op\_type == SPEOpType\_OtherSVE then
    op_class = '00';
    op_subclass = '0'::evl::'1'::pred::fp::'0';
  elseif SPESampleOpAttr.op\_type == SPEOpType\_OthersME then
    op_class = '00';
    op_subclass = '1'::ets[3:1]::'1'::ets[0]::fp::'0';
  elseif SPESampleOpAttr.op\_type == SPEOpType\_Branch then
    op_class = '10';
    var cr : bits(2) = '00';
    var gcs : bit = '0';
    if IsFeatureImplemented(FEAT_SPE_CRR) then
      if SPESampleOpAttr.branch\_has\_link then
        cr = '01';
      elseif SPESampleOpAttr.procedure\_return then
        cr = '10';
      else
        cr = '11';
      end;
    end;
    if IsFeatureImplemented(FEAT_GCS) then
      if (SPESampleOpAttr.ldst\_type == SPELDSTType\_GCS &&
        (SPESampleOpAttr.branch\_has\_link || SPESampleOpAttr.procedure\_return)) then
        gcs = '1';
      end;
    end;
    op_subclass = Zeros{3}::cr::gcs::indirect::cond;
  elseif SPESampleOpAttr.op\_type IN {SPEOpType\_Load, SPEOpType\_Store, SPEOpType\_LoadAtomic} then
    op_class = '01';
    case SPESampleOpAttr.ldst\_type of
      when SPELDSTType\_NV2 => op_subclass = '0011000'::ldst;
      when SPELDSTType\_Extended => op_subclass = '000'::ar::excl::at::'1'::ldst;
      when SPELDSTType\_General => op_subclass = Zeros{7}::ldst;
      when SPELDSTType\_SIMDFP => op_subclass = '0000010'::ldst;
      when SPELDSTType\_SVESME => op_subclass = sg::evl::'1'::pred::'0'::ldst;
      when SPELDSTType\_Unspecified => op_subclass = '0001000'::ldst;
      when SPELDSTType\_Tags => op_subclass = '0001010'::ldst;
      when SPELDSTType\_MemCopy => op_subclass = '0010000'::ldst;
      when SPELDSTType\_MemSet => op_subclass = '00100101';
      when SPELDSTType\_GCS => op_subclass = '01000'::common::'0'::ldst;
      when SPELDSTType\_GCSSS2 => unreachable;
      // GCSSS2 is converted to GCS, should not appear here
      otherwise => unreachable;
    end;
  else
    unreachable;
  end;
  return (op_class, op_subclass);
end;
```



```

// SPEConstructRecord()
// =====
// Create new record and populate it with packets using sample storage data.
// This is an example implementation, packets may appear in
// any order as long as the record ends with an End or Timestamp packet.

func SPEConstructRecord()
begin
    // Empty the record.
    SPEEmptyRecord();

    var op_class : bits(2);
    var op_subclass : bits(8);
    (op_class, op_subclass) = SPEConstructClass();
    // Add operation details
    SPEAddPacketToRecord{8}('01', '10'::op_class, op_subclass);

    // Add contextEL1 if available
    if SPESampleContextEL1Valid then
        SPEAddPacketToRecord{32}('01', '0100', SPESampleContextEL1);
    end;

    // Add contextEL2 if available
    if SPESampleContextEL2Valid then
        SPEAddPacketToRecord{32}('01', '0101', SPESampleContextEL2);
    end;

    // Add valid counters
    for counter_index = 0 to (SPEMaxCounters - 1) do
        if SPESampleCounterValid[[counter_index]] then
            if counter_index >= 8 then
                // Need extended format
                SPEAddByteToRecord('001000'::counter_index[4:3]);
            end;
            // Check for overflow
            let large_counters : boolean = ImpDefBool("SPE 16bit counters");
            if large_counters then
                if SPESampleCounter[[counter_index]] > 0xFFFF then
                    SPESampleCounter[[counter_index]] = 0xFFFF;
                end;
            else
                if SPESampleCounter[[counter_index]] > 0xFFF then
                    SPESampleCounter[[counter_index]] = 0xFFF;
                end;
            end;

            // Add byte0 for short format (byte1 for extended format)
            SPEAddPacketToRecord{16}('10', '1'::counter_index[2:0],
                                     SPESampleCounter[[counter_index]][15:0]);
        end;
    end;

    // Add valid addresses
    if IsFeatureImplemented(FEAT_SPE_PBT) then
        // Under the some conditions, it is IMPLEMENTATION_DEFINED whether
        // previous branch packet is present.
        let include_prev_br : boolean = (ImpDefBool(
            "SPE get prev br if not br"));
        if SPESampleOpAttr.op_type != SPEOpType_Branch && !include_prev_br then
            SPESampleAddressValid[[SPEAddrPosPrevBranchTarget]] = FALSE;
        end;
    end;

    // Data Virtual address should not be collected if this was an NV2 access and Statistical
    // Profiling is disabled at EL2.
    if !StatisticalProfilingEnabled(EL2) && SPESampleOpAttr.ldst_type == SPELDSTType_NV2 then
        SPESampleAddressValid[[SPEAddrPosDataVirtual]] = FALSE;
    end;

    for address_index = 0 to (SPEMaxAddrs - 1) do
        if SPESampleAddressValid[[address_index]] then
            if address_index >= 8 then
                // Need extended format
                SPEAddByteToRecord('001000'::address_index[4:3]);
            end;
            // Add byte0 for short format (byte1 for extended format)
            SPEAddPacketToRecord{64}('10', '0'::address_index[2:0],
                                     SPESampleAddress[[address_index]]);
        end;
    end;

    // Add Data Source
    if SPESampleDataSourceValid then
        let ds_payload_size : integer{} = SPEGetDataSourcePayloadSize();
        SPEAddPacketToRecord{8 * ds_payload_size}('01', '0011',
                                                    SPESampleDataSource[8*ds_payload_size-1:0]);
    end;
end;

```

```

// Add events
// Get size of payload in bytes.
let payload_size : integer{} = SPEGetEventsPayloadSize();
SPEAddPacketToRecord{8 * payload_size}('01', '0010', SPESampleEvents[8*payload_size-1:0]);

// Add Timestamp to end the record if one is available.
// Otherwise end with an End packet.
if SPESampleTimestampValid then
    SPEAddPacketToRecord{64}('01', '0001', SPESampleTimestamp);
else
    SPEAddByteToRecord('00000001');
end;

// Add padding
while SPERecordSize MOD (1<<UInt(PMBIDR_EL1().Align)) != 0 looplimit 2048 do
    SPEAddByteToRecord(Zeros{8});
end;
SPEWriteToBuffer();
CTI_SignalEvent(CrossTriggerIn SPESample);
end;

```

Library pseudocode for aarch64/debug/statisticalprofiling/SPECycle

```

// SPECycle()
// =====
// Function called at the end of every cycle. Responsible for asserting interrupts
// and advancing counters.

func SPECycle()
begin
    if !IsFeatureImplemented(FEAT_SPE) then
        return;
    end;

    // Increment pending counters
    if SPESampleInFlight then
        for i = 0 to (SPEMaxCounters - 1) do
            if SPESampleCounterPending[[i]] then
                SPESampleCounter[[i]] = SPESampleCounter[[i]] + 1;
            end;
        end;
    end;

    // Assert PMBIRQ if appropriate.
    if SPEInterruptEnabled() && PMBSR_EL1().S == '1' then
        SetInterruptRequestLevel(InterruptID PMBIRQ, HIGH);
    else
        SetInterruptRequestLevel(InterruptID PMBIRQ, LOW);
    end;
end;

```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEEmptyRecord

```

// SPEEmptyRecord()
// =====
// Reset record data.

func SPEEmptyRecord()
begin
    SPERecordSize = 0;
    for i = 0 to (SPEMaxRecordSize - 1) do
        SPERecordData[[i]] = Zeros{8};
    end;
end;

```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEEncodeETS

```
// SPEEncodeETS()
// =====
// Encodes an integer tile size length into the ets field for the SPE operation type packet.

func SPEEncodeETS(size : integer) => bits(4)
begin
    var ets : bits(4);
    if size <= 128 then
        ets = '0000';
    elsif size <= 256 then
        ets = '0001';
    elsif size <= 512 then
        ets = '0010';
    elsif size <= 1024 then
        ets = '0011';
    elsif size <= 2048 then
        ets = '0100';
    elsif size <= 4096 then
        ets = '0101';
    elsif size <= 8192 then
        ets = '0110';
    elsif size <= 16384 then
        ets = '0111';
    elsif size <= 32768 then
        ets = '1000';
    elsif size <= 65536 then
        ets = '1001';
    elsif size <= 131072 then
        ets = '1010';
    elsif size <= 262144 then
        ets = '1011';
    else
        ets = '1111';
    end;
    return ets;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEEncodeEVL

```
// SPEEncodeEVL()
// =====
// Encodes an integer vector length into the evl field for the SPE operation type packet.

func SPEEncodeEVL(vl : integer) => bits(3)
begin
    var evl : bits(3);
    if vl <= 32 then
        evl = '000';
    elsif vl <= 64 then
        evl = '001';
    elsif vl <= 128 then
        evl = '010';
    elsif vl <= 256 then
        evl = '011';
    elsif vl <= 512 then
        evl = '100';
    elsif vl <= 1024 then
        evl = '101';
    elsif vl <= 2048 then
        evl = '110';
    else
        if IsFeatureImplemented(FEAT_SPE_SME) then
            evl = '111';
        else
            unreachable;
        end;
    end;
    return evl;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEEvent

```
// SPEEvent()
// =====
// Called when a PMU event is generated by the sampled structure.
// Sets appropriate bit in SPESampleStorage.events.

func SPEEvent(pmuevent : bits(16))
begin
    if !SPESampleInFlight then
        return;
    end;
    case pmuevent of
        when PMU_EVENT_DSNP_HIT_RD =>
            if IsFeatureImplemented(FEAT_SPEvlp4) then
                SPESampleEvents[23] = '1'; // Snoop hit
            end;
        when {PMU_EVENT_L1D_LFB_HIT_RD, PMU_EVENT_L2D_LFB_HIT_RD,
            PMU_EVENT_L3D_LFB_HIT_RD, PMU_EVENT_LL_LFB_HIT_RD} =>
            if IsFeatureImplemented(FEAT_SPEvlp4) then
                SPESampleEvents[22] = '1'; // Recent fetch
            end;
        when {PMU_EVENT_L1D_CACHE_HITM_RD, PMU_EVENT_L2D_CACHE_HITM_RD,
            PMU_EVENT_L3D_CACHE_HITM_RD, PMU_EVENT_LL_CACHE_HITM_RD} =>
            if IsFeatureImplemented(FEAT_SPEvlp4) then
                SPESampleEvents[21] = '1'; // Modified
            end;
        when PMU_EVENT_L2D_CACHE_LMISS_RD =>
            if IsFeatureImplemented(FEAT_SPEvlp4) then
                SPESampleEvents[20] = '1'; // L2 miss
            end;
        when PMU_EVENT_L2D_CACHE_RD =>
            if IsFeatureImplemented(FEAT_SPEvlp4) then
                SPESampleEvents[19] = '1'; // L2 access
            end;
        when PMU_EVENT_SVE_PRED_EMPTY_SPEC =>
            if IsFeatureImplemented(FEAT_SPEvlp1) then
                SPESampleEvents[18] = '1'; // Empty predicate
            end;
        when PMU_EVENT_SVE_PRED_NOT_FULL_SPEC =>
            if IsFeatureImplemented(FEAT_SPEvlp1) then
                SPESampleEvents[17] = '1'; // Partial predicate
            end;
        when PMU_EVENT_LDST_ALIGN_LAT =>
            if IsFeatureImplemented(FEAT_SPEvlp1) then
                SPESampleEvents[11] = '1'; // Misaligned
            end;
        when PMU_EVENT_REMOTE_ACCESS => SPESampleEvents[10] = '1'; // Remote access
        when PMU_EVENT_LL_CACHE_MISS => SPESampleEvents[9] = '1'; // LLC miss
        when PMU_EVENT_LL_CACHE => SPESampleEvents[8] = '1'; // LLC access
        when PMU_EVENT_BR_MIS_PRED => SPESampleEvents[7] = '1'; // Mispredicted
        when PMU_EVENT_BR_MIS_PRED_RETIRE => SPESampleEvents[7] = '1'; // Not taken
        when PMU_EVENT_DTLB_WALK => SPESampleEvents[5] = '1'; // TLB walk
        when PMU_EVENT_L1D_TLB => SPESampleEvents[4] = '1'; // TLB access
        when PMU_EVENT_L1D_CACHE_REFILL =>
            if !IsFeatureImplemented(FEAT_SPEvlp4) then
                SPESampleEvents[3] = '1'; // L1 refill
            end;
        when PMU_EVENT_L1D_CACHE_LMISS_RD =>
            if IsFeatureImplemented(FEAT_SPEvlp4) then
                SPESampleEvents[3] = '1'; // L1 miss
            end;
        when PMU_EVENT_L1D_CACHE => SPESampleEvents[2] = '1'; // L1 access
        when PMU_EVENT_INST_RETIRE => SPESampleEvents[1] = '1'; // Retire
        when PMU_EVENT_EXC_TAKEN => SPESampleEvents[0] = '1'; // Exception
        otherwise => return;
    end;
    return;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEFilterByDataSource

```
// SPEFilterByDataSource()
// =====
// Carry out filtering by data source.

func SPEFilterByDataSource(optype : SPEOpType) => boolean
begin
    // Filtering by Data Source
    var is_rejected_data_source : boolean = FALSE;
    if (IsFeatureImplemented(FEAT_SPE_FDS) && SPESampleDataSourceValid &&
        (optype IN {SPEOpType\_Load, SPEOpType\_LoadAtomic})) then
        let data_source : bits(16) = SPESampleDataSource;
        let index : integer         = UInt(data_source[5:0]);
        let is_ds : boolean         = PMSDSFR_EL1()[index] == '1';
        if is_ds then PMUEvent(PMU_EVENT_SAMPLE_FEED_DS); end;
        if PMSFCR_EL1().FDS == '1' then
            // Filtering by Data Source is enabled
            is_rejected_data_source = !is_ds;
        end;
    end;
    return is_rejected_data_source;
end;
```



```

// SPEFilterByEvents()
// =====
// Carries out filtering by the event bits.

func SPEFilterByEvents(events : bits(64)) => (boolean, boolean)
begin
    // "mask" defines which Events packet bits are checked by the filter
    var mask : bits(64) = Zeros{};
    let impdef_mask : bits(64) = ImpDefBits{}("SPE mask");

    mask[63:48] = impdef_mask[63:48];
    if IsFeatureImplemented(FEAT_SPE_SME) && IsFeatureImplemented(FEAT_SPEvlp4) then
        mask[25:24] = '11'; // Streaming mode, SMCU
    end;
    if IsFeatureImplemented(FEAT_SPEvlp4) then
        mask[23:19] = '11111'; // Snooper hit, recent fetch, modified,
        // L2 miss, L2 access
    else
        mask[31:24] = impdef_mask[31:24];
    end;
    if IsFeatureImplemented(FEAT_SPEvlp1) && IsFeatureImplemented(FEAT_SVE) then
        mask[18:17] = '11'; // Predicates
    end;
    mask[15:12] = impdef_mask[15:12];
    if IsFeatureImplemented(FEAT_SPEvlp1) then
        mask[11] = '1'; // Data alignment
    end;
    if IsFeatureImplemented(FEAT_SPEvlp4) then
        mask[10:8] = '111'; // Remote access, LLC access, LLC miss
    else
        mask[10:8] = impdef_mask[10:8];
    end;
    mask[7] = '1'; // Mispredicted
    if IsFeatureImplemented(FEAT_SPE_FnE) then
        mask[6] = '1'; // Not taken
    end;
    mask[5,3,1] = '111'; // TLB walk, L1 miss, retired
    if IsFeatureImplemented(FEAT_SPEvlp4) then
        mask[4,2] = '11'; // TLB access, L1 access
    else
        mask[4,2] = impdef_mask[4,2];
    end;

    let e : bits(64) = events AND mask;

    // Filtering by event
    let evfr : bits(64) = PMSEVFR_EL1() AND mask;
    var is_rejected_event : boolean = FALSE;
    let is_evt : boolean = IsZero(NOT(e) AND evfr);
    if PMSFCR_EL1().FE == '1' then
        // Filtering by event is enabled
        if !IsZero(evfr) then
            // Not a CONSTRAINED UNPREDICTABLE case
            is_rejected_event = !is_evt;
        else
            is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        end;
    end;

    // Filtering by inverse event
    var is_rejected_nevent : boolean = FALSE;
    var is_nevt : boolean;
    if IsFeatureImplemented(FEAT_SPE_FnE) then
        let nevfr : bits(64) = PMSNEVFR_EL1() AND mask;
        is_nevt = IsZero(e AND nevfr);
        if PMSFCR_EL1().FnE == '1' then
            // Inverse filtering by event is enabled
            if !IsZero(nevfr) then
                // Not a CONSTRAINED UNPREDICTABLE case
                is_rejected_nevent = !is_nevt;
            else
                is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
            end;
        end;
    else
        is_nevt = TRUE; // not implemented
    end;
    if is_evt && is_nevt then
        PMUEvent(PMU_EVENT_SAMPLE_FEED_EVENT);
    end;
    if (IsFeatureImplemented(FEAT_SPE_FnE) && PMSFCR_EL1().[FnE,FE]== '11' &&
        !IsZero(PMSEVFR_EL1() AND PMSNEVFR_EL1() AND mask)) then
        // CONSTRAINED UNPREDICTABLE case due to combination of filter and inverse filter
        is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
    end;

    return (is_rejected_event, is_rejected_nevent);
end;

```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEFilterByLatency

```
// SPEFilterByLatency()
// =====
// Carries out filtering by latency.

func SPEFilterByLatency(total_latency : integer) => boolean
begin

    let is_lat : boolean = (total_latency >= UInt(PMSLATFR_EL1().MINLAT));
    if is_lat then PMUEvent(PMU_EVENT_SAMPLE_FEED_LAT); end;

    var is_rejected_latency : boolean = FALSE;
    if PMSFCR_EL1().FL == '1' then
        // Filtering by latency is enabled
        if !IsZero(PMSLATFR_EL1().MINLAT) then
            // Not a CONSTRAINED UNPREDICTABLE case
            is_rejected_latency = is_rejected_latency || !is_lat;
        else
            is_rejected_latency = ConstrainUnpredictableBool(Unpredictable\_BADPMSFCR);
        end;
    end;

    return is_rejected_latency;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEFilterByType

```
// SPEFilterByType()
// =====
// Returns TRUE if the operation is to be discarded because of its OpAttrs, and FALSE otherwise.

func SPEFilterByType(opattr : SPEOpAttr) => boolean
begin
    // Bit positions in the PMSFCR_EL1().[TYPE, TYPEm] fields
    let B : integer = 0;
    let LD : integer = 1;
    let ST : integer = 2;
    let FP : integer = 3;
    let SIMD : integer = 4;

    var flags : bits(5) = Zeros{};
    var ctrl : bits(5) = Zeros{};
    // With GCS, Branch Link and Procedure Return instructions write and read the GCS.
    // BL and RET instructions with GCS enabled call SPESampleLoadStore() before
    // branch/packet construction, setting ldst_type to indicate a GCS access.
    let is_gcs_ldst : boolean = (opattr.op_type == SPEOpType Branch &&
        opattr.ldst_type == SPELDSTType GCS);
    let is_load : boolean = (opattr.op_type IN {SPEOpType Load, SPEOpType LoadAtomic} ||
        (is_gcs_ldst && opattr.procedure_return));
    let is_store : boolean = (opattr.op_type IN {SPEOpType Store, SPEOpType LoadAtomic} ||
        (is_gcs_ldst && opattr.branch_has_link));
    flags[B] = (if opattr.op_type == SPEOpType Branch then '1' else '0');
    flags[LD] = (if is_load then '1' else '0');
    flags[ST] = (if is_store then '1' else '0');
    flags[FP] = (if opattr.is_floating_point then '1' else '0');
    flags[SIMD] = (if opattr.is_simd then '1' else '0');
    ctrl[2:0] = PMSFCR_EL1().TYPE[2:0];
    var mask : bits(5) = Zeros{};
    if IsFeatureImplemented(FEAT_SPE_EFT) then
        ctrl[4:3] = PMSFCR_EL1().TYPE[4:3];
        mask[4:0] = PMSFCR_EL1().TYPEm;
    end;

    let ctrl_or : bits(5) = (ctrl AND (NOT mask));
    let ctrl_and : bits(5) = (ctrl AND mask);

    let is_op : boolean = ((IsZero(ctrl_or) || !IsZero(flags AND ctrl_or)) &&
        ((flags AND mask) == ctrl_and));

    if flags[B] == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_BR); end;
    if flags[LD] == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_LD); end;
    if flags[ST] == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_ST); end;
    if flags[FP] == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_FP); end;
    if flags[SIMD] == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_SIMD); end;

    var is_rejected_type : boolean = FALSE;
    if PMSFCR_EL1().FT == '1' then
        // Filtering by type is enabled
        if IsFeatureImplemented(FEAT_SPE_EFT) || !IsZero(ctrl) then
            is_rejected_type = !is_op;
        else
            is_rejected_type = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        end;
    end;
    if is_op && (!IsZero(ctrl) || !IsZero(mask)) then PMUEvent(PMU_EVENT_SAMPLE_FEED_OP); end;

    return is_rejected_type;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEFreezeOnEvent

```
// SPEFreezeOnEvent()
// =====
// Returns TRUE if PMU event counter idx should be frozen due to an SPE event, and FALSE otherwise.

func SPEFreezeOnEvent(idx : integer) => boolean
begin
    let counters : integer = NUM_PMU_COUNTERS;
    assert (idx >= 0 &&
            (idx < counters || idx == CYCLE\_COUNTER\_ID ||
             (idx == INSTRUCTION\_COUNTER\_ID && IsFeatureImplemented(FEAT_PMUv3_ICNTR))));

    if !IsFeatureImplemented(FEAT_SPEv1p2) || !IsFeatureImplemented(FEAT_PMUv3p7) then
        return FALSE;
    end;
    if idx == CYCLE\_COUNTER\_ID && !IsFeatureImplemented(FEAT_SPE_DPFZS) then
        // FZS does not affect the cycle counter when FEAT_SPE_DPFZS is not implemented
        return FALSE;
    end;

    let freeze_on_event : boolean = PMBLIMITR_EL1().[E,PMFZ] == '11';
    let stopped : boolean = SPEProfilingStopped();

    case GetPMUCounterRange(idx) of
        when PMUCounterRange\_R1 =>
            return freeze_on_event && stopped && PMCR_EL0().FZS == '1';
        when PMUCounterRange\_R2 =>
            return freeze_on_event && stopped && MDCR_EL2().HPMFZS == '1';
        otherwise =>
            return FALSE;
    end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetDataSource

```
// SPEGetDataSource()
// =====
// Returns a tuple indicating the data source for an access passed to SPESampleLoadStore, and
// whether the data source is valid.

impdef func SPEGetDataSource(is_load : boolean,
                             accdesc : AccessDescriptor,
                             addrdesc : AddressDescriptor) => (boolean, bits(16))

begin
    return (TRUE, Zeros{16});
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetDataSourcePayloadSize

```
// SPEGetDataSourcePayloadSize()
// =====
// Returns the size of the Data Source payload in bytes.

func SPEGetDataSourcePayloadSize() => integer{1..2}
begin
    return ImpDefInt("SPE Data Source packet payload size") as integer{1..2};
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetEventsPayloadSize

```
// SPEGetEventsPayloadSize()
// =====
// Returns the size in bytes of the Events packet payload as an integer.

func SPEGetEventsPayloadSize() => integer{1..4}
begin
    return ImpDefInt("SPE Events packet payload size") as integer{1..4};
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetRandomBoolean

```
// SPEGetRandomBoolean()
// =====
// Returns a random or pseudo-random boolean value.

impdef func SPEGetRandomBoolean() => boolean
begin
    return SPEGetRandomInterval() [0] == '1';
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetRandomInterval

```
// SPEGetRandomInterval()
// =====
// Returns a random or pseudo-random byte for resetting COUNT or ECOUNT.

impdef func SPEGetRandomInterval() => bits(8)
begin
    return \_PC[0+:8];
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEISB

```
// SPEISB()
// =====
// Called by ISB instruction, correctly calls SPEBranch to save previous branches.

func SPEISB()
begin
    let address : bits(64)          = PC64() + 4;
    let branch_type : BranchType    = BranchType DIR;
    let branch_conditional : boolean = FALSE;
    let taken : boolean              = FALSE;
    let is_isb : boolean             = TRUE;

    SPEBranch{64}(address, branch_type, branch_conditional, taken, is_isb);
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEInterruptEnabled

```
// SPEInterruptEnabled()
// =====
// Return TRUE if the SPE interrupt request (PMBIRQ) is enabled, FALSE otherwise.

func SPEInterruptEnabled() => boolean
begin
    return EffectivePMSCR\_EL1\_EE() == '00';
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPELDSTType

```
// SPELDSTType
// =====
// Type of a load or store operation.

type SPELDSTType of enumeration {
    SPELDSTType_NV2,
    SPELDSTType_Extended,
    SPELDSTType_General,
    SPELDSTType_SIMDFP,
    SPELDSTType_SVESME,
    SPELDSTType_Tags,
    SPELDSTType_MemCopy,
    SPELDSTType_MemSet,
    SPELDSTType_GCS,
    SPELDSTType_GCSSS2,
    SPELDSTType_Unspecified
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEMultiAccessSample

```
// SPEMultiAccessSample()
// =====
// Called by instructions which make at least one store and one load access, where the configuration
// of the operation type filter affects which access is sampled.

func SPEMultiAccessSample() => boolean
begin
    // If loads or stores are filtered out, the other should be recorded.
    // If neither or both are filtered out, pick one in an unbiased way.

    // Bit positions in the PMSFCR_EL1().[TYPE, TYPEm] fields.
    let LD : integer = 1;
    let ST : integer = 2;

    // Are loads allowed by filter?
    let loads_pass_filter : boolean = PMSFCR_EL1().FT == '1' && PMSFCR_EL1().TYPE[LD] == '1';
    // Are stores allowed by filter?
    let stores_pass_filter : boolean = PMSFCR_EL1().FT == '1' && PMSFCR_EL1().TYPE[ST] == '1';

    var record_load : boolean;
    if loads_pass_filter && !stores_pass_filter then
        // Only loads pass filter
        record_load = TRUE;
    elseif !loads_pass_filter && stores_pass_filter then
        // Only stores pass filter
        record_load = FALSE;
    else
        // Both loads and stores pass the filter or neither pass the filter.
        // Pick between the load or the store access (pseudo-)randomly.
        record_load = SPEGetRandomBoolean\(\);
    end;
    return record_load;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEOpAttr

```
// SPEOpAttr
// =====
// Attributes of sampled operation filtered by SPECollectRecord().

type SPEOpAttr of record {
    op_type : SPEOpType,
    ldst_type : SPELDSTType,
    branch_is_direct : boolean,
    branch_has_link : boolean,
    procedure_return : boolean,
    is_conditional : boolean,
    is_floating_point : boolean,
    is_simd : boolean,
    cond_pass : boolean,
    at : boolean,
    is_acquire_release : boolean,
    is_predicated : boolean,
    evl : bits(3),
    is_gather_scatter : boolean,
    is_exclusive : boolean,
    ets : bits(4),
    addr_valid : boolean
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEOpType

```
// SPEOpType
// =====
// Types of operation filtered by SPECollectRecord().

type SPEOpType of enumeration {
    SPEOpType_Load,           // Any memory-read operation other than atomics, compare-and-swap,
                                // and swap
    SPEOpType_Store,          // Any memory-write operation, including atomics without return
    SPEOpType_LoadAtomic,     // Atomics with return, compare-and-swap and swap
    SPEOpType_Branch,         // Software write to the PC
    SPEOpType_OtherSVE,       // Other SVE operation
    SPEOpType_OtherSME,       // Other SME operation
    SPEOpType_Other,          // Any other class of operation
    SPEOpType_Invalid
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEProfilingStopped

```
// SPEProfilingStopped()
// =====

readonly func SPEProfilingStopped() => boolean
begin
    var stopped : boolean = (PMBSR_EL1().S == '1');
    if IsFeatureImplemented(FEAT_SPE_EXC) then
        if HaveEL\(EL3\) && MDCR_EL3().PMSEE == '1x' then
            stopped = stopped || (PMBSR_EL3().S == '1');
        end;
        if EffectivePMSCR\_EL2\_EE\(\) == '1x' then
            stopped = stopped || (PMBSR_EL2().S == '1');
        end;
    end;
    return stopped;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEResetSampleCounter

```
// SPEResetSampleCounter()
// =====
// Reset PMSICR_EL1.Counter

func SPEResetSampleCounter()
begin
    PMSICR_EL1().COUNT[31:8] = PMSIRR_EL1().INTERVAL;
    if PMSIRR_EL1().RND == '1' && !IsFeatureImplemented(FEAT_SPE_ERnd) then
        PMSICR_EL1().COUNT[7:0] = SPEGetRandomInterval\(\);
    else
        PMSICR_EL1().COUNT[7:0] = Zeros{8};
    end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEResetSampleStorage

```
// SPEResetSampleStorage()
// =====
// Reset all variables inside sample storage.

func SPEResetSampleStorage()
begin
    // Context values
    SPESampleContextEL1 = Zeros{32};
    SPESampleContextEL1Valid = FALSE;
    SPESampleContextEL2 = Zeros{32};
    SPESampleContextEL2Valid = FALSE;

    // Counter values
    for i = 0 to (SPEMaxCounters - 1) do
        SPESampleCounter[[i]] = 0;
        SPESampleCounterValid[[i]] = FALSE;
        SPESampleCounterPending[[i]] = FALSE;
    end;

    // Address values
    for i = 0 to (SPEMaxAddrs - 1) do
        SPESampleAddressValid[[i]] = FALSE;
        SPESampleAddress[[i]] = Zeros{64};
    end;

    // Data source values
    SPESampleDataSource = Zeros{16};
    SPESampleDataSourceValid = FALSE;

    // Timestamp values
    SPESampleTimestamp = Zeros{64};
    SPESampleTimestampValid = FALSE;

    // Event values
    SPESampleEvents = Zeros{64};

    // Operation attributes
    SPESampleOpAttr.op_type = SPEOpType_Invalid;
    SPESampleOpAttr.ldst_type = SPELDSTType_Unspecified;
    SPESampleOpAttr.branch_is_direct = FALSE;
    SPESampleOpAttr.branch_has_link = FALSE;
    SPESampleOpAttr.procedure_return = FALSE;
    SPESampleOpAttr.is_conditional = FALSE;
    SPESampleOpAttr.is_floating_point = FALSE;
    SPESampleOpAttr.is_simd = FALSE;
    SPESampleOpAttr.cond_pass = FALSE;
    SPESampleOpAttr.at = FALSE;
    SPESampleOpAttr.is_acquire_release = FALSE;
    SPESampleOpAttr.is_exclusive = FALSE;
    SPESampleOpAttr.is_predicated = FALSE;
    SPESampleOpAttr.evl = '000';
    SPESampleOpAttr.is_gather_scatter = FALSE;
    SPESampleOpAttr.addr_valid = FALSE;
end;

var SPERecordData : array [[SPEMaxRecordSize]] of bits(8);
var SPERecordSize : integer;
```


Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddAddressPCVirtual

```
// SPESampleAddAddressPCVirtual()
// =====
// Save the current PC address to sample storage.

func SPESampleAddAddressPCVirtual()
begin
    var ns : bit;
    var nse : bit;
    case CurrentSecurityState() of
        when SS\_Secure =>
            ns = '0';
            nse = '0';
        when SS\_NonSecure =>
            ns = '1';
            nse = '0';
        when SS\_Realm =>
            ns = '1';
            nse = '1';
        otherwise => unreachable;
    end;
    let el : bits(2) = PSTATE.EL;

    let pc : bits(64) = ThisInstrAddr{()};
    SPESampleAddress[[SPEAddrPosPCVirtual]][55:0] = pc[55:0];
    SPESampleAddress[[SPEAddrPosPCVirtual]][63:56] = ns::el::nse::Zeros{4};
    SPESampleAddressValid[[SPEAddrPosPCVirtual]] = TRUE;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddContext

```
// SPESampleAddContext()
// =====
// Save contexts to sample storage if appropriate.

func SPESampleAddContext()
begin
    if CollectContextIDR1() then
        SPESampleContextEL1 = CONTEXTIDR\_EL1() [31:0];
        SPESampleContextEL1Valid = TRUE;
    end;
    if CollectContextIDR2() then
        SPESampleContextEL2 = CONTEXTIDR\_EL2() [31:0];
        SPESampleContextEL2Valid = TRUE;
    end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddTimeStamp

```
// SPESampleAddTimeStamp()
// =====
// Save the appropriate type of timestamp to sample storage.

func SPESampleAddTimeStamp()
begin
    let timestamp : TimeStamp = CollectTimeStamp();
    case timestamp of
        when TimeStamp\_None =>
            SPESampleTimestampValid = FALSE;
        otherwise =>
            SPESampleTimestampValid = TRUE;
            SPESampleTimestamp = GetTimestamp(timestamp);
    end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleCollision

```
// SPESampleCollision()
// =====
// Called when there is an SPE sample collision.

func SPESampleCollision()
begin
    // Sample collision with the previous sample
    PMUEvent(PMU_EVENT_SAMPLE_COLLISION);
    let target_el : bits(2) = DefaultSPEEvent();
    PMBSR\_EL(target_el).COLL = '1';
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleExtendedLoadStore

```
// SPESampleExtendedLoadStore()
// =====
// Sets the subclass of the operation type packet for
// extended load/store operations.

func SPESampleExtendedLoadStore(ar : boolean, excl : boolean, at : boolean, is_load : boolean)
begin
    SPESampleOpAttr.is_acquire_release = ar;
    SPESampleOpAttr.is_exclusive       = excl;
    SPESampleOpAttr.ldst_type          = SPELDSTType_Extended;
    SPESampleOpAttr.at                 = at;
    if is_load then
        if at then
            SPESampleOpAttr.op_type = SPEOpType_LoadAtomic;
        else
            SPESampleOpAttr.op_type = SPEOpType_Load;
        end;
    else
        SPESampleOpAttr.op_type = SPEOpType_Store;
    end;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleGCSSS2

```
// SPESampleGCSSS2()
// =====
// Sets the subclass of the operation type packet for GCSSS2 load/store operations.

func SPESampleGCSSS2()
begin
    // GCSSS2 does a read and a write.
    let record_load : boolean = SPEMultiAccessSample();
    SPESampleOpAttr.op_type = if record_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_GCSSS2;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleGeneralPurposeLoadStore

```
// SPESampleGeneralPurposeLoadStore()
// =====
// Sets the subclass of the operation type packet for general
// purpose load/store operations.

func SPESampleGeneralPurposeLoadStore(is_load : boolean)
begin
    SPESampleOpAttr.ldst_type = SPELDSTType_General;
    SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleLoadStore

```
// SPESampleLoadStore()
// =====
// Called if a sample is in flight when writing or reading memory,
// indicating that the operation being sampled is in the Load, Store or atomic category.

func SPESampleLoadStore(is_load : boolean, accdesc : AccessDescriptor,
                        addrdesc : AddressDescriptor)
begin
    // Check if this access type is suitable to be sampled.
    // This implementation of SPE always samples the first access made by a suitable instruction.
    // FEAT_MOPS instructions are an exception, where the first load or first store access may be
    // selected based on the configuration of the sample filters.
    if accdesc.acctype IN {AccessType\_SPE,
                          AccessType\_IFETCH,
                          AccessType\_TRBE,
                          AccessType\_DC,
                          AccessType\_TTW,
                          AccessType\_AT} then
        return;
    end;

    var sample_access : boolean = FALSE;
    // For FEAT_MOPS and FEAT_GCS GCSSS2 instructions which perform both loads and stores, the
    // filter configuration will influence which part of the access is chosen to be sampled.
    if (SPESampleOpAttr.ldst_type IN
        {SPELDSTType\_MemCopy, SPELDSTType\_MemSet, SPELDSTType\_GCSSS2}) then
        // SPEMultiAccessSample() will have been called before this function,
        // and chooses whether to sample a load or a store.
        var sample_load : boolean;
        sample_load = SPESampleOpAttr.op_type IN {SPEOpType\_Load, SPEOpType\_LoadAtomic};

        // If no valid data has been collected, and this operation is acceptable for sampling.
        if !SPESampleOpAttr.addr_valid && (is_load == sample_load) then
            sample_access = TRUE;
        end;
    else
        if !SPESampleOpAttr.addr_valid then
            sample_access = TRUE;
        end;
    end;

    if sample_access then
        // Data access virtual address
        SPESetDataVirtualAddress(addrdesc.vaddress);

        // Data access physical address
        if CollectPhysicalAddress() then
            SPESetDataPhysicalAddress(addrdesc, accdesc);
        end;

        SPESampleOpAttr.addr_valid = TRUE;
    end;

    if SPESampleOpAttr.op_type == SPEOpType\_Invalid then
        // Set as unspecified load/store by default, instructions will overwrite this if it does not
        // apply to them.
        SPESampleOpAttr.op_type = if is_load then SPEOpType\_Load else SPEOpType\_Store;

        if accdesc.acctype == AccessType\_NV2 then
            // NV2 register load/store
            SPESampleOpAttr.ldst_type = SPELDSTType\_NV2;
        end;
    end;

    // Set SPELDSTType to GCS for all GCS instruction, overwriting type GCSSS2.
    // After selection of which operation of a GCSSS2 instruction to sample, GCSSS2 is treated the
    // same as other GCS instructions.
    if accdesc.acctype == AccessType\_GCS then
        SPESampleOpAttr.ldst_type = SPELDSTType\_GCS;
        // If the GCS access is from a BL or RET, this will get overwritten to SPEOpType_Branch.
        SPESampleOpAttr.op_type = if is_load then SPEOpType\_Load else SPEOpType\_Store;
    end;

    (SPESampleDataSourceValid, SPESampleDataSource) = SPEGetDataSource(is_load, accdesc, addrdesc);
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleMemCopy

```
// SPESampleMemCopy()
// =====
// Sets the subclass of the operation type packet for Memory Copy load/store
// operations.

func SPESampleMemCopy()
begin
    // MemCopy does a read and a write.
    let record_load : boolean = SPEMultiAccessSample\(\);
    SPESampleOpAttr.op_type = if record_load then SPEOpType\_Load else SPEOpType\_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType\_MemCopy;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleMemSet

```
// SPESampleMemSet()
// =====
// Callback used by memory set instructions to pass data back to the SPU.

func SPESampleMemSet()
begin
    SPESampleOpAttr.op_type = SPEOpType\_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType\_MemSet;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOnSharedResource

```
// SPESampleOnSharedResource()
// =====
// Called when the sampled instruction is executed on an SMCU or other shared resource, sets bit 25
// of the events packet to 0b1.
func SPESampleOnSharedResource()
begin
    SPESampleEvents[25] = '1';
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOpOther

```
// SPESampleOpOther()
// =====
// Add other operation to sample storage.

func SPESampleOpOther(conditional : boolean, cond_pass : boolean,
    is_fp : boolean, is_simd : boolean)
begin
    SPESampleOpAttr.is_simd = is_simd;
    SPESampleOpOther(conditional, cond_pass, is_fp);
end;

func SPESampleOpOther(conditional : boolean, cond_pass : boolean, is_fp : boolean)
begin
    SPESampleOpAttr.cond_pass = cond_pass;
    SPESampleOpAttr.is_floating_point = is_fp;
    SPESampleOpOther(conditional);
end;

func SPESampleOpOther(conditional : boolean)
begin
    SPESampleOpAttr.is_conditional = conditional;
    SPESampleOpAttr.op_type = SPEOpType\_Other;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOpSMEArrayOther

```
// SPESampleOpSMEArrayOther()
// =====
// Sets the subclass of the operation type packet to Other, SME array.

func SPESampleOpSMEArrayOther(floating_point : boolean, size : integer)
begin
    // If the sampled effective vector or tile size is not a power of two, or is less than 128 bits,
    // the value is rounded up before it is encoded in the ets field.

    SPESampleOpAttr.is_floating_point = floating_point;
    SPESampleOpAttr.ets = SPEEncodeETS(size);
    SPESampleOpAttr.op_type = SPEOpType\_OtherSME;
    SPESampleOpAttr.is_simd = TRUE;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOpSVEOther

```
// SPESampleOpSVEOther()
// =====
// Callback used by SVE, Other operations to pass data back to the SPU.

func SPESampleOpSVEOther(vl : integer, predicated : boolean, floating_point : boolean)
begin
    SPESampleOpAttr.is_predicated      = predicated;
    SPESampleOpAttr.is_floating_point = floating_point;
    SPESampleOpAttr.evl                = SPEEncodeEVL(vl);
    SPESampleOpAttr.op_type            = SPEOpType_OtherSVE;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOpSVESMELoadStore

```
// SPESampleOpSVESMELoadStore()
// =====
// Callback used by SVE or SME loads and stores to pass data to SPE.

func SPESampleOpSVESMELoadStore(is_gather_scatter : boolean, vl : integer, predicated : boolean,
                                is_load : boolean)
begin
    SPESampleOpAttr.is_gather_scatter = is_gather_scatter;
    SPESampleOpAttr.is_predicated     = predicated;
    SPESampleOpAttr.evl               = SPEEncodeEVL(vl);
    assert SPESampleOpAttr.evl != '111';
    SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_SVESME;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleSIMDFPLoadStore

```
// SPESampleSIMDFPLoadStore()
// =====
// Sets the subclass of the operation type packet for SIMD & FP
// load store operations.

func SPESampleSIMDFPLoadStore(is_load : boolean, scalar : boolean)
begin
    SPESampleOpAttr.ldst_type = SPELDSTType_SIMDFP;
    SPESampleOpAttr.op_type   = if is_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.is_simd   = !scalar;
    // Scalar operations in SIMD&FP are treated as floating point.
    SPESampleOpAttr.is_floating_point = scalar;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESetDataPhysicalAddress

```
// SPESetDataPhysicalAddress()
// =====
// Called from SampleLoadStore() to save data physical packet.

func SPESetDataPhysicalAddress(addrdesc : AddressDescriptor, accdesc : AccessDescriptor)
begin
    var ns : bit;
    var nse : bit;
    var ch : bit;
    var lat : bits(4);
    case addrdesc.paddress.paspace of
        when PAS\_Secure =>
            ns = '0';
            nse = '0';
        when PAS\_NonSecure =>
            ns = '1';
            nse = '0';
        when PAS\_Realm =>
            ns = '1';
            nse = '1';
        otherwise => unreachable;
    end;

    if IsFeatureImplemented(FEAT_MTE2) then
        if accdesc.tagchecked then
            ch = '1';
            lat = AArch64\_LogicalAddressTag(addrdesc.vaddress);
        else
            ch = '0';
            // If the access is Unchecked, this is an IMPLEMENTATION_DEFINED choice
            // between 0b0000 and the Logical Address Tag
            var zero_unchecked : boolean;
            zero_unchecked = ImpDefBool("SPE PAT for tag unchecked access zero");
            if !zero_unchecked then
                lat = AArch64\_LogicalAddressTag(addrdesc.vaddress);
            else
                lat = Zeros{4};
            end;
        end;
    else
        ch = '0';
        lat = '0000';
    end;

    let paddr : bits(NUM_PABITS) = addrdesc.paddress.address;
    SPESampleAddress[[SPEAddrPosDataPhysical]][55:0] = ZeroExtend{56}(paddr);
    SPESampleAddress[[SPEAddrPosDataPhysical]][63:56] = ns::ch::'0'::nse::lat;
    SPESampleAddressValid[[SPEAddrPosDataPhysical]] = TRUE;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESetDataVirtualAddress

```
// SPESetDataVirtualAddress()
// =====
// Called from SampleLoadStore() to save data virtual packet.
// Also used by exclusive load/stores to save virtual addresses if exclusive monitor is lost
// before a read/write is completed.

func SPESetDataVirtualAddress(vaddress : bits(64))
begin
    var tbi : bit;
    tbi = EffectiveTBI(vaddress, FALSE, PSTATE.EL);
    var non_tbi_is_zeros : boolean;
    non_tbi_is_zeros = ImpDefBool("SPE non-tbi tag is zero");
    if tbi == '1' || !non_tbi_is_zeros then
        SPESampleAddress[[SPEAddrPosDataVirtual]][63:0] = vaddress[63:0];
    else
        SPESampleAddress[[SPEAddrPosDataVirtual]][55:0] = vaddress[55:0];
        SPESampleAddress[[SPEAddrPosDataVirtual]][63:56] = Zeros{8};
    end;
    SPESampleAddressValid[[SPEAddrPosDataVirtual]] = TRUE;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESetCounter

```
// SPESetCounter()
// =====
// Enables incrementing of the counter at the passed index when SPECycle is called.

func SPESetCounter(counter_index : integer)
begin
    assert counter_index < SPEMaxCounters;
    SPESampleCounterPending[[counter_index]] = TRUE;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESStartSample

```
// SPESStartSample()
// =====
// Called to start the sampling process. Returns TRUE if a new sample is to be collected, and
// FALSE otherwise.

func SPESStartSample() => boolean
begin
    if !StatisticalProfilingEnabled() then
        return FALSE;
    end;
    PMUEvent(PMU_EVENT_SAMPLE_POP);
    if !SPEToCollectSample() then
        return FALSE;
    end;

    if SPESampleInFlight then
        SPESampleCollision();
        return FALSE;
    end;

    SPESampleInFlight = TRUE;

    SPESStartCounter(SPECcounterPosTotalLatency);
    SPESStartCounter(SPECcounterPosIssueLatency);
    SPESampleAddAddressPCVirtual();

    // Many operations are type other and not conditional, can save footprint
    // and overhead by having this as the default and not calling SPESampleOpOther
    // if conditional == FALSE
    SPESampleOpOther(FALSE);

    SPESampleAddContext();

    // Timestamp may be collected at any point in the sampling operation.
    // Collecting prior to execution is one possible choice.
    // This choice is IMPLEMENTATION DEFINED.
    SPESampleAddTimeStamp();

    return TRUE;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESStopCounter

```
// SPESStopCounter()
// =====
// Disables incrementing of the counter at the passed index when SPECycle is called.

func SPESStopCounter(counter_index : integer)
begin
    SPESampleCounterValid[counter_index] = TRUE;
    SPESampleCounterPending[counter_index] = FALSE;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEToCollectSample

```
// SPEToCollectSample()
// =====
// Returns TRUE if the instruction which is about the executed should be sampled, and FALSE
// otherwise.

func SPEToCollectSample() => boolean
begin
    if IsZero(PMSICR_EL1().COUNT) then
        SPEResetSampleCounter();
    else
        PMSICR_EL1().COUNT = PMSICR_EL1().COUNT - 1;
        if IsZero(PMSICR_EL1().COUNT) then
            if PMSIRR_EL1().RND == '1' && IsFeatureImplemented(FEAT_SPE_ERnd) then
                PMSICR_EL1().ECOUNT = SPEGetRandomInterval();
                return IsZero(PMSICR_EL1().ECOUNT);
            else
                return TRUE;
            end;
        end;
    end;
    if (PMSIRR_EL1().RND == '1' && IsFeatureImplemented(FEAT_SPE_ERnd) &&
        !IsZero(PMSICR_EL1().ECOUNT)) then
        PMSICR_EL1().ECOUNT = PMSICR_EL1().ECOUNT - 1;
        if IsZero(PMSICR_EL1().ECOUNT) then
            return TRUE;
        end;
    end;
    return FALSE;
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEWriteToBuffer

```
// SPEWriteToBuffer()
// =====
// Write the active record to the Profiling Buffer.

func SPEWriteToBuffer()
begin
    assert ProfilingBufferEnabled() && !SPEProfilingStopped();

    // Check alignment
    let align : integer{} = UInt(PMBIDR_EL1().Align);
    let aligned : boolean = IsAlignedP2(PMBPTR_EL1().PTR, align);

    let ttw_abort_as_fault : boolean = (ImpDefBool(
        "Report SPE ExtAbort on TTW as fault"));

    var owning_ss : SecurityState;
    var owning_el : bits(2);
    (owning_ss, owning_el) = ProfilingBufferOwner();
    let accdesc : AccessDescriptor = CreateAccDescSPE(owning_ss, owning_el);

    let start_vaddr : bits(64) = PMBPTR_EL1();
    for i = 0 to SPERecordSize - 1 do
        // If a previous write did not cause an issue
        if !SPEProfilingStopped() then
            let address : bits(64) = PMBPTR_EL1();
            var memstatus : PhysMemRetStatus;
            var addrdesc : AddressDescriptor;
            (memstatus, addrdesc) = DebugMemWrite(address, accdesc, aligned,
                SPERecordData[[i]]);

            let fault : FaultRecord = addrdesc.fault;
            let ttw_abort : boolean = fault.statuscode IN {Fault\_SyncExternalOnWalk,
                Fault\_SyncParityOnWalk};

            if IsFault(fault.statuscode) && (!ttw_abort || ttw_abort_as_fault) then
                DebugWriteFault(address, fault);
            elseif IsFault(memstatus) || (ttw_abort && !ttw_abort_as_fault) then
                DebugWriteExternalAbort(memstatus, addrdesc, start_vaddr);
            end;

            // Move pointer if no Buffer Management Event has been caused.
            if !SPEProfilingStopped() then
                PMBPTR_EL1() = PMBPTR_EL1() + 1;
            end;
        end;
    end;

    return;
end;
```


Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =====
// Return TRUE if Statistical Profiling is Enabled in the current EL, FALSE otherwise.

func StatisticalProfilingEnabled() => boolean
begin
    return StatisticalProfilingEnabled\(PSTATE.EL\);
end;

// StatisticalProfilingEnabled()
// =====
// Return TRUE if Statistical Profiling is Enabled in the specified EL, FALSE otherwise.

func StatisticalProfilingEnabled(el : bits(2)) => boolean
begin
    if (!IsFeatureImplemented(FEAT_SPE) || UsingAArch32\(\) ||
        !ProfilingBufferEnabled\(\) || SPEProfilingStopped\(\) || Halted\(\)) then
        return FALSE;
    end;

    let tge_set : boolean = EL2Enabled\(\) && HCR_EL2().TGE == '1';
    let (owning_ss, owning_el) : (SecurityState, bits(2)) = ProfilingBufferOwner\(\);
    if (UInt(owning_el) < UInt(el) || (tge_set && owning_el == EL1) ||
        owning_ss != SecurityStateAtEL(el)) then
        return FALSE;
    end;
    var spe_bit : bit;
    case el of
        when EL3 => unreachable;
        when EL2 => spe_bit = PMSCR_EL2().E2SPE;
        when EL1 => spe_bit = PMSCR_EL1().E1SPE;
        when EL0 => spe_bit = (if tge_set then PMSCR_EL2().E0HSPE else PMSCR_EL1().E0SPE);
    end;

    return spe_bit == '1';
end;
```

Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```
// TimeStamp
// =====

type TimeStamp of enumeration {
    TimeStamp_None,           // No timestamp
    TimeStamp_CoreSight,      // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,       // Physical counter value with no offset
    TimeStamp_OffsetPhysical, // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual };      // Physical counter value minus CNTVOFF_EL2
```

Library pseudocode for aarch64/debug/takeexceptiondbg/AArch64_TakeExceptionInDebugState

```
// AArch64_TakeExceptionInDebugState()
// =====
// Take an exception in Debug state to an Exception level using AArch64.

noreturn func AArch64_TakeExceptionInDebugState(target_el : bits(2), exception_in : ExceptionRecord)
begin
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    assert target_el != EL3 || EDSCR().SDD == '0';
    var except : ExceptionRecord = exception_in;

    if !IsFeatureImplemented(FEAT_ExS) || SCTLR\_EL(target_el).EIS == '1' then
        // Synchronize the context, including Instruction Fetch Barrier effect
        SynchronizeContext();
    end;

    // If coming from AArch32 state, the top parts of the X() registers might be set to zero
    let from_32 : boolean = UsingAArch32();
    if from_32 then AArch64\_MaybeZeroRegisterUppers(); end;
    if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);
    end;

    AArch64\_ReportException(except, target_el);

    if IsFeatureImplemented(FEAT_GCS) then
        PSTATE.EXLOCK = '0'; // Effective value of GCSCR_ELx.EXLOCKEN is 0 in Debug state
    end;

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR\_ELx() = ARBITRARY : bits(64);
    ELR\_ELx() = ARBITRARY : bits(64);
    // PSTATE.[SS,D,A,I,F] are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.[SS,D,A,I,F] = ARBITRARY : bits(5);
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
    if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = ARBITRARY : bit; end;
    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1'; end;
    PSTATE.IL = '0';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    if (IsFeatureImplemented(FEAT_PAN) && (PSTATE.EL == EL1 ||
        (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR\_ELx().SPAN == '0') then
        PSTATE.PAN = '1';
    end;

    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0'; // PSTATE.J is RES0
    end;
    if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTTYPE = '00'; end;
    DLR_EL0() = ARBITRARY : bits(64);
    DSPSR_EL0() = ARBITRARY : bits(64);

    EDSCR().ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR PE state flags.

    EndOfInstruction(EndOfInstructionReason\_Exception);
end;
```

Library pseudocode for aarch64/debug/watchpoint/AArch64_WatchpointByteMatch

```
// AArch64_WatchpointByteMatch()
// =====

func AArch64_WatchpointByteMatch(n : integer, vaddress : bits(64)) => boolean
begin
    let dbgtop : AddressSize = DebugAddrTop();
    let cmpbottom : integer{} = if DBGWVR_EL1(n)[2] == '1' then 2 else 3; // Word or doubleword
    var bottom : integer{} = cmpbottom;
    let select : integer{} = UInt(vaddress[cmpbottom-1:0]);
    var byte_select_match : boolean = (DBGWCR_EL1(n).BAS[select] != '0');
    var mask : integer{} = UInt(DBGWCR_EL1(n).MASK);

    // If DBGWCR_EL1(n).MASK is a nonzero value and DBGWCR_EL1(n).BAS is not set to '11111111', or
    // DBGWCR_EL1(n).BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1(n).BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable WPMASKANDBAS);
    else
        let LSB : bits(8) = (DBGWCR_EL1(n).BAS AND NOT(DBGWCR_EL1(n).BAS - 1));
        let MSB : bits(8) = (DBGWCR_EL1(n).BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable WPBASCONTIGUOUS);
            bottom = 3; // For the whole doubleword
        end;
    end;

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        var c : Constraint;
        var unpred_mask : integer;
        (c, unpred_mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable RESWPMASK);
        assert c IN {Constraint DISABLED, Constraint NONE, Constraint UNKNOWN};
        case c of
            when Constraint DISABLED => return FALSE; // Disabled
            when Constraint NONE => mask = 0; // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value
            otherwise => mask = unpred_mask as integer{3..31};
        end;
    end;

    // When FEAT_LVA3 is not implemented, if the DBGWVR_EL1(n).RESS field bits are not a
    // sign extension of the MSB of DBGWVR_EL1(n).VA, it is UNPREDICTABLE whether they
    // appear to be included in the match.
    let unpredictable_ress : boolean = (dbgtop < 55 && !IsOnes(DBGWVR_EL1(n)[63:dbgtop]) &&
        !IsZero(DBGWVR_EL1(n)[63:dbgtop]) &&
        ConstrainUnpredictableBool(Unpredictable DBGxVR_RESS));

    let cmpmsb : integer{} = if unpredictable_ress then 63 else dbgtop;
    let cmpls : integer{} = if mask > bottom then mask else bottom;
    let bottombit : integer{} = bottom;
    var WVR_match : boolean = (vaddress[cmpmsb:cmpls] == DBGWVR_EL1(n)[cmpmsb:cmpls]);
    if mask > bottom then
        // If masked bits of DBGWVR_EL1(n) are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1(n)[cmpls-1:bottombit]) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable WPMASKEDBITS);
        end;
    end;

    return (WVR_match && byte_select_match);
end;
```

Library pseudocode for aarch64/debug/watchpoint/AArch64_WatchpointMatch

```
// AArch64_WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.

func AArch64_WatchpointMatch(n : integer, vaddress : bits(64), size : integer,
                             accdesc : AccessDescriptor) => WatchpointInfo
begin
    assert !ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert n < NumWatchpointsImplemented\(\);

    let enabled : boolean      = IsWatchpointEnabled(n);
    let linked : boolean      = DBGWCR_EL1(n).WT == '1';
    let isbreakpnt : boolean   = FALSE;
    let lbnx : bits(2) = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGWCR_EL1(n).LBNX else '00';
    let linked_n : integer{} = UInt(lbnx :: DBGWCR_EL1(n).LBN);
    let ssce : bit            = if IsFeatureImplemented(FEAT_RME) then DBGWCR_EL1(n).SSCE else '0';
    let mismatch : boolean     = IsFeatureImplemented(FEAT_BWE2) && DBGWCR_EL1(n).WT2 == '1';
    let state_match : boolean = AArch64\_StateMatch(DBGWCR_EL1(n).SSC, ssce, DBGWCR_EL1(n).HMC,
                                                  DBGWCR_EL1(n).PAC,
                                                  linked, linked_n, isbreakpnt, PC64\(\), accdesc);

    var watchptinfo : WatchpointInfo;
    var ls_match : boolean;
    case DBGWCR_EL1(n).LSC[1:0] of
        when '00' => ls_match = FALSE;
        when '01' => ls_match = accdesc.read;
        when '10' => ls_match = accdesc.write || accdesc.acctype == AccessType\_DC;
        when '11' => ls_match = TRUE;
    end;

    var value_match : boolean = FALSE;
    watchptinfo.vaddress = vaddress;
    for byte = 0 to size - 1 do
        if (!value_match &&
            !AddressInNaturallyAlignedBlock(watchptinfo.vaddress, vaddress + byte)) then
            // Watchpoint should report an address which is in
            // the naturally aligned block of the matched address.
            watchptinfo.vaddress = vaddress + byte;
        end;
        value_match = value_match || AArch64\_WatchpointByteMatch(n, vaddress + byte);
    end;

    watchptinfo.watchpt_num = n;
    watchptinfo.value_match = value_match;
    if !(state_match && ls_match && enabled) then
        watchptinfo.wptype = WatchpointType\_Inactive;
        watchptinfo.value_match = FALSE;
    elsif mismatch then
        watchptinfo.wptype = WatchpointType\_AddrMismatch;
    else
        watchptinfo.wptype = WatchpointType\_AddrMatch;
    end;
    return watchptinfo;
end;
```

Library pseudocode for aarch64/debug/watchpoint/IsWatchpointEnabled

```
// IsWatchpointEnabled()
// =====
// Returns TRUE if the effective value of DBGWCR_EL1(n).E is '1', and FALSE otherwise.

func IsWatchpointEnabled(n : integer) => boolean
begin
    if (n > 15 &&
        ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEEnabled()) ||
         (HaltOnBreakpointOrWatchpoint() && EDSCR2().EHBWE == '0')) then
        return FALSE;
    end;
    return DBGWCR_EL1(n).E == '1';
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_Abort

```
// AArch64_Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime.

func AArch64_Abort(fault : FaultRecord)
begin
    if IsDebugException(fault) then
        if fault.accessdesc.acctype == AccessType IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException\_VectorCatch then
                AArch64\_VectorCatchException(fault);
            else
                AArch64\_BreakpointException(fault);
            end;
        else
            AArch64\_WatchpointException(fault);
        end;
    elseif fault.gpcf.gpf != GPCF\_None && ReportAsGPCEXception(fault) then
        TakeGPCEXception(fault);
    elseif fault.statuscode == Fault\_TagCheck then
        AArch64\_RaiseTagCheckFault(fault);
    elseif IsExternalAbort(fault) && !IsExternalSyncAbort(fault) then
        PendSErrorInterrupt(fault);
    elseif fault.accessdesc.acctype == AccessType IFETCH then
        AArch64\_InstructionAbort(fault);
    else
        AArch64\_DataAbort(fault);
    end;
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_AbortSyndrome

```
// AArch64_AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.
func AArch64_AbortSyndrome(excepttype : Exception, fault : FaultRecord,
    target_el : bits(2)) => ExceptionRecord
begin
    var except : ExceptionRecord = ExceptionSyndrome(excepttype);

    except.syndrome = AArch64\_FaultSyndrome(excepttype, fault, target_el);
    if excepttype IN {Exception\_NV2Watchpoint, Exception\_Watchpoint} then
        except.vaddress = fault.watchptinfo.vaddress;
    elseif fault.statuscode == Fault\_TagCheck then
        if IsFeatureImplemented(FEAT_MTE_TAGGED_FAR) then
            except.vaddress = ZeroExtend{64}(fault.vaddress);
        else
            except.vaddress = ARBITRARY : bits(4) :: fault.vaddress[59:0];
        end;
    else
        except.vaddress = ZeroExtend{64}(fault.vaddress);
    end;

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS\_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;
    end;

    if except.syndrome.iss[14] == '1' then
        except.pavalid = TRUE;
        except.paddress = fault.paddress;
    else
        except.pavalid = FALSE;
        except.paddress = ARBITRARY : FullAddress;
    end;

    return except;
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_CheckPCAlignment

```
// AArch64_CheckPCAlignment()
// =====

func AArch64_CheckPCAlignment()
begin
    let pc : bits(64) = ThisInstrAddr{}();

    if pc[1:0] != '00' then
        AArch64\_PCAlignmentFault();
    end;
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_DataAbort

```
// AArch64_DataAbort()
// =====

func AArch64_DataAbort(fault : FaultRecord)
begin
    var target_el : bits(2);
    if IsExternalAbort(fault) then
        target_el = SyncExternalAbortTarget(fault);
    else
        let route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
            (HCR_EL2().TGE == '1' ||
             (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF\_None &&
              HCR_EL2().GPF == '1') ||
             (IsFeatureImplemented(FEAT_NV2) &&
              fault.accessdesc.acctype == AccessType\_NV2) ||
              IsSecondStage(fault)));

        if PSTATE.EL == EL3 then
            target_el = EL3;
        elsif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;
        end;
    end;

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let route_to_serr : boolean = (IsExternalAbort(fault) &&
        AArch64\_RouteToSErrorOffset(target_el));
    let vect_offset : integer = if route_to_serr then 0x180 else 0x0;

    var except : ExceptionRecord;
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType\_NV2 then
        except = AArch64\_AbortSyndrome(Exception\_NV2DataAbort, fault, target_el);
    else
        except = AArch64\_AbortSyndrome(Exception\_DataAbort, fault, target_el);
    end;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_EffectiveTCF

```
// AArch64_EffectiveTCF()
// =====
// Indicate if a Tag Check Fault should cause a synchronous exception,
// be asynchronously accumulated, or have no effect on the PE.

func AArch64_EffectiveTCF(el : bits(2), read : boolean) => TCFType
begin
    var tcf : bits(2);

    let regime : Regime = TranslationRegime(el);

    case regime of
        when Regime\_EL3 => tcf = SCTLR_EL3().TCF;
        when Regime\_EL2 => tcf = SCTLR_EL2().TCF;
        when Regime\_EL20 => tcf = if el == EL0 then SCTLR_EL2().TCF0 else SCTLR_EL2().TCF;
        when Regime\_EL10 => tcf = if el == EL0 then SCTLR_EL1().TCF0 else SCTLR_EL1().TCF;
        otherwise => unreachable;
    end;

    if tcf == '11' then // Reserved value
        if !IsFeatureImplemented(FEAT_MTE_ASYM_FAULT) then
            (-,tcf) = ConstrainUnpredictableBits{2}(Unpredictable\_RESTCF);
        end;
    end;

    case tcf of
        when '00' => // Tag Check Faults have no effect on the PE
            return TCFType\_Ignore;
        when '01' => // Tag Check Faults cause a synchronous exception
            return TCFType\_Sync;
        when '10' =>
            if IsFeatureImplemented(FEAT_MTE_ASYNC) then
                // If asynchronous faults are implemented,
                // Tag Check Faults are asynchronously accumulated
                return TCFType\_Async;
            else
                // Otherwise, Tag Check Faults have no effect on the PE
                return TCFType\_Ignore;
            end;
        when '11' =>
            if IsFeatureImplemented(FEAT_MTE_ASYM_FAULT) then
                // Tag Check Faults cause a synchronous exception on reads or on
                // a read/write access, and are asynchronously accumulated on writes
                if read then
                    return TCFType\_Sync;
                else
                    return TCFType\_Async;
                end;
            else
                // Otherwise, Tag Check Faults have no effect on the PE
                return TCFType\_Ignore;
            end;
        otherwise =>
            unreachable;
    end;
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_InstructionAbort

```
// AArch64_InstructionAbort()
// =====

func AArch64_InstructionAbort(fault : FaultRecord)
begin
    // External aborts on instruction fetch must be taken synchronously
    if IsFeatureImplemented(FEAT_DoubleFault) then
        assert fault.statuscode != Fault\_AsyncExternal;
    end;

    var target_el : bits(2);
    if IsExternalAbort(fault) then
        target_el = SyncExternalAbortTarget(fault);
    else
        let route_to_el2 : boolean = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
            (HCR_EL2().TGE == '1' ||
            (IsFeatureImplemented(FEAT_RME) &&
            fault.gpcf.gpf != GPCF\_None &&
            HCR_EL2().GPF == '1') ||
            IsSecondStage(fault)));

        if PSTATE.EL == EL3 then
            target_el = EL3;
        elseif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;
        end;
    end;

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var vect_offset : integer;

    if IsExternalAbort(fault) && AArch64\_RouteToErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    end;

    let except : ExceptionRecord = AArch64\_AbortSyndrome(Exception\_InstructionAbort, fault,
        target_el);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_PCAlignmentFault

```
// AArch64_PCAlignmentFault()
// =====
// Called on unaligned program counter in AArch64 state.

func AArch64_PCAlignmentFault()
begin

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception\_PCAlignment);
    except.vaddress = ThisInstrAddr{64}();
    var target_el : bits(2) = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elseif EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    end;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```


Library pseudocode for aarch64/exceptions/aborts/AArch64_RaiseTagCheckFault

```
// AArch64_RaiseTagCheckFault()
// =====
// Raise a Tag Check Fault exception.

func AArch64_RaiseTagCheckFault(fault : FaultRecord)
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
    var target_el : bits(2) = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elseif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    end;

    let except : ExceptionRecord = AArch64\_AbortSyndrome(Exception\_DataAbort, fault, target_el);
    AArch64\_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_ReportTagCheckFault

```
// AArch64_ReportTagCheckFault()
// =====
// Records a Tag Check Fault exception into the appropriate TFSR_ELx.

func AArch64_ReportTagCheckFault(el : bits(2), ttbr : bit)
begin
    case el of
        when EL3 => assert ttbr == '0'; TFSR_EL3().TF0 = '1';
        when EL2 => if ttbr == '0' then TFSR_EL2().TF0 = '1'; else TFSR_EL2().TF1 = '1'; end;
        when EL1 => if ttbr == '0' then TFSR_EL1().TF0 = '1'; else TFSR_EL1().TF1 = '1'; end;
        when EL0 => if ttbr == '0' then TFSR_EL0().TF0 = '1'; else TFSR_EL0().TF1 = '1'; end;
    end;
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_RouteToErrorOffset

```
// AArch64_RouteToErrorOffset()
// =====
// Returns TRUE if synchronous External abort exceptions are taken to the
// appropriate SError vector offset, and FALSE otherwise.

func AArch64_RouteToErrorOffset(target_el : bits(2)) => boolean
begin
    if !IsFeatureImplemented(FEAT_DoubleFault) then return FALSE; end;

    var ease_bit : bit;
    case target_el of
        when EL3 =>
            ease_bit = SCR_EL3().EASE;
        when EL2 =>
            if IsFeatureImplemented(FEAT_DoubleFault2) && IsSCTLR2EL2Enabled() then
                ease_bit = SCTLR2_EL2().EASE;
            else
                ease_bit = '0';
            end;
        when EL1 =>
            if IsFeatureImplemented(FEAT_DoubleFault2) && IsSCTLR2EL1Enabled() then
                ease_bit = SCTLR2_EL1().EASE;
            else
                ease_bit = '0';
            end;
    end;
    return (ease_bit == '1');
end;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64_SPAlignmentFault

```
// AArch64_SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.

func AArch64_SPAlignmentFault()
begin

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    let except : ExceptionRecord = ExceptionSyndrome(Exception\_SPAlignment);

    var target_el : bits(2) = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    end;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

```
// BranchTargetException()
// =====
// Raise branch target exception.

func AArch64_BranchTargetException(vaddress : bits(52))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception\_BranchTarget);
    except.syndrome.iss[1:0] = PSTATE.BTYPE;
    except.syndrome.iss[24:2] = Zeros{23}; // RES0

    var target_el : bits(2) = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    end;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/aborts/TCFType

```
// TCFType
// =====

type TCFType of enumeration { TCFType_Sync, TCFType_Async, TCFType_Ignore };
```

Library pseudocode for aarch64/exceptions/aborts/TakeGPCEException

```
// TakeGPCEException()
// =====
// Report Granule Protection Exception faults

func TakeGPCEException(fault : FaultRecord)
begin
    assert IsFeatureImplemented(FEAT_RME);
    assert IsFeatureImplemented(FEAT_LSE);
    assert IsFeatureImplemented(FEAT_HAFDBS);
    assert IsFeatureImplemented(FEAT_DoubleFault);

    var except : ExceptionRecord;

    except.exceptype = Exception\_GPC;
    except.vaddress = ZeroExtend{64}(fault.vaddress);
    except.paddress = fault.paddress;
    except.pavalid = TRUE;

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS\_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;
    end;

    except.syndrome.iss2[11] = if fault.hdbssf then '1' else '0'; // HDBSSF
    if fault.accessdesc.acctype == AccessType\_GCS then
        except.syndrome.iss2[8] = '1'; //GCS
    end;

    // Populate the fields grouped in ISS
    except.syndrome.iss[24:22] = Zeros{3}; // RES0
    except.syndrome.iss[21] = if fault.gpcfs2walk then '1' else '0'; // S2PTW
    if fault.accessdesc.acctype == AccessType\_IFETCH then
        except.syndrome.iss[20] = '1'; // InD
    else
        except.syndrome.iss[20] = '0'; // InD
    end;
    except.syndrome.iss[19:14] = EncodeGPCSC(fault.gpcf); // GPCSC
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType\_NV2 then
        except.syndrome.iss[13] = '1'; // VNCR
    else
        except.syndrome.iss[13] = '0'; // VNCR
    end;
    except.syndrome.iss[12:11] = '00'; // RES0
    except.syndrome.iss[10:9] = '00'; // RES0

    if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
        except.syndrome.iss[8] = '1'; // CM
    else
        except.syndrome.iss[8] = '0'; // CM
    end;

    except.syndrome.iss[7] = if fault.s2fslwalk then '1' else '0'; // S1PTW

    if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
        except.syndrome.iss[6] = '1'; // WnR
    elseif fault.statuscode IN {Fault\_UnsupportedAtomicHWUpdate, Fault\_Exclusive} then
        except.syndrome.iss[6] = ARBITRARY : bit; // WnR
    elseif fault.accessdesc.atomicop && IsExternalAbort(fault) then
        except.syndrome.iss[6] = ARBITRARY : bit; // WnR
    else
        except.syndrome.iss[6] = if fault.write then '1' else '0'; // WnR
    end;

    except.syndrome.iss[5:0] = EncodeLDFSC(fault.statuscode, fault.level); // xFSC

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let target_el : bits(2) = EL3;

    var vect_offset : integer;
    if IsExternalAbort(fault) && AArch64\_RouteToErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    end;

    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/async/AArch64_TakeDelegatedSErrorException

```
// AArch64_TakeDelegatedSErrorException()
// =====

func AArch64_TakeDelegatedSErrorException()
begin
    assert IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 && SCR_EL3().[EnDSE,DSE] == '11';

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset = 0x180;
    var except : ExceptionRecord = ExceptionSyndrome(Exception_SError);

    var target_el : bits(2);
    var dsei_masked : boolean;
    (dsei_masked, target_el) = AArch64_DelegatedSErrorTarget();
    assert !dsei_masked;
    except.syndrome.iss[24] = VSESR_EL3().IDS;
    except.syndrome.iss[23:0] = VSESR_EL3().ISS;
    ClearPendingDelegatedSError();

    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/async/AArch64_TakePhysicalFIQException

```
// AArch64_TakePhysicalFIQException()
// =====

func AArch64_TakePhysicalFIQException()
begin
    let route_to_el3 : boolean = EffectiveSCR_EL3_FIQ() == '1';
    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2().TGE == '1' || HCR_EL2().FMO == '1'));
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x100;
    let except : ExceptionRecord = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/exceptions/async/AArch64_TakePhysicalIRQException

```
// AArch64_TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

func AArch64_TakePhysicalIRQException()
begin
    let route_to_el3 : boolean = EffectiveSCR_EL3_IRQ() == '1';
    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2().TGE == '1' || HCR_EL2().IMO == '1'));
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x80;

    let except : ExceptionRecord = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/exceptions/async/AArch64_TakePhysicalSErrorException

```
// AArch64_TakePhysicalSErrorException()
// =====

func AArch64_TakePhysicalSErrorException(implicit_esb : boolean)
begin
    var masked : boolean;
    var target_el : bits(2);

    (masked, target_el) = PhysicalErrorTarget();
    assert !masked;

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x180;
    let fault : FaultRecord = GetPendingPhysicalSError();
    var except : ExceptionRecord = ExceptionSyndrome(Exception\_SError);
    let is_esb : boolean = FALSE;
    let syndrome : bits(25) = AArch64\_PhysicalErrorSyndrome(is_esb, implicit_esb);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    end;

    if except.syndrome.iss[14] == '1' then
        except.pavalid = TRUE;
        except.paddress = fault.paddress;
    else
        except.pavalid = FALSE;
        except.paddress = ARBITRARY : FullAddress;
    end;
    except.syndrome.iss = syndrome;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/async/AArch64_TakeVirtualFIQException

```
// AArch64_TakeVirtualFIQException()
// =====

func AArch64_TakeVirtualFIQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    // Virtual IRQ enabled if TGE==0 and FMO==1
    assert HCR\_EL2\(\).TGE == '0' && HCR\_EL2\(\).FMO == '1';

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x100;

    let except : ExceptionRecord = ExceptionSyndrome(Exception\_FIQ);

    AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/async/AArch64_TakeVirtualIRQException

```
// AArch64_TakeVirtualIRQException()
// =====

func AArch64_TakeVirtualIRQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    // Virtual IRQ enabled if TGE==0 and IMO==1
    assert HCR\_EL2\(\).TGE == '0' && HCR\_EL2\(\).IMO == '1';

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x80;

    let except : ExceptionRecord = ExceptionSyndrome(Exception\_IRQ);

    AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/async/AArch64_TakeVirtualSErrorException

```
// AArch64_TakeVirtualSErrorException()
// =====

func AArch64_TakeVirtualSErrorException()
begin

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    // Virtual SError enabled if TGE==0 and AMO==1 or TMEA==1
    assert (HCR\_EL2\(\).TGE == '0' &&
           (HCR\_EL2\(\).AMO == '1' || (IsHCRXEL2Enabled() && HCRX\_EL2\(\).TMEA == '1')));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x180;
    var except : ExceptionRecord = ExceptionSyndrome(Exception\_SError);

    if IsFeatureImplemented(FEAT_RAS) then
        except.syndrome.iss[24] = VSESR\_EL2\(\).IDS;
        except.syndrome.iss[23:0] = VSESR\_EL2\(\).ISS;
    else
        let syndrome : bits(25) = ImpDefBits{}("Virtual SError syndrome");
        let impdef_syndrome : boolean = syndrome[24] == '1';
        if impdef_syndrome then except.syndrome.iss = syndrome; end;
    end;

    ClearPendingVirtualSError();
    AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/debug/AArch64_BreakpointException

```
// AArch64_BreakpointException()
// =====

func AArch64_BreakpointException(fault : FaultRecord)
begin
    assert PSTATE.EL != EL3;

    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                (HCR\_EL2\(\).TGE == '1' || MDCR\_EL2\(\).TDE == '1'));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var target_el : bits(2);
    let vect_offset : integer = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    let vaddress : bits(64) = ARBITRARY : bits(64);
    let except : ExceptionRecord = AArch64\_AbortSyndrome(Exception\_Breakpoint, fault, target_el);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/debug/AArch64_SoftwareBreakpoint

```
// AArch64_SoftwareBreakpoint()
// =====

func AArch64_SoftwareBreakpoint(immediate : bits(16))
begin

    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} &&
                                EL2Enabled() && (HCR\_EL2\(\).TGE == '1' || MDCR\_EL2\(\).TDE == '1'));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception\_SoftwareBreakpoint);
    except.syndrome.iss[15:0] = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/exceptions/debug/AArch64_SoftwareStepException

```
// AArch64_SoftwareStepException()
// =====

func AArch64_SoftwareStepException()
begin
    assert PSTATE.EL != EL3;

    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                   (HCR\_EL2().TGE == '1' || MDCR\_EL2().TDE == '1'));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception\_SoftwareStep);
    if SoftwareStep\_DidNotStep() then
        except.syndrome.iss[24] = '0';
    else
        except.syndrome.iss[24] = '1';
        except.syndrome.iss[6] = if SoftwareStep\_SteppedEX() then '1' else '0';
    end;
    except.syndrome.iss[5:0] = '100010'; // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/exceptions/debug/AArch64_VectorCatchException

```
// AArch64_VectorCatchException()
// =====
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

func AArch64_VectorCatchException(fault : FaultRecord)
begin
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR\_EL2().TGE == '1' || MDCR\_EL2().TDE == '1');

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    let vaddress : bits(64) = ARBITRARY : bits(64);
    let except : ExceptionRecord = AArch64\_AbortSyndrome(Exception\_VectorCatch, fault, EL2);

    AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/debug/AArch64_WatchpointException

```
// AArch64_WatchpointException()
// =====

func AArch64_WatchpointException(fault : FaultRecord)
begin
    assert PSTATE.EL != EL3;

    let route_to_el2 : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                                   (HCR\_EL2().TGE == '1' || MDCR\_EL2().TDE == '1'));

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var target_el : bits(2);
    let vect_offset : integer = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    var except : ExceptionRecord;
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType\_NV2 then
        except = AArch64\_AbortSyndrome(Exception\_NV2Watchpoint, fault, target_el);
    else
        except = AArch64\_AbortSyndrome(Exception\_Watchpoint, fault, target_el);
    end;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64_ExceptionClass

```
// AArch64_ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in ESR

func AArch64_ExceptionClass(exceptype : Exception, target_el : bits(2)) => (integer,bit)
begin
    var il_is_valid : boolean = TRUE;
    let from_32 : boolean = UsingAArch32();
    var ec : integer;
    case exceptype of
        when Exception Uncategorized => ec = 0x00; il_is_valid = FALSE;
        when Exception WFXTrap => ec = 0x01;
        when Exception CP15RRTTrap => ec = 0x03; assert from_32;
        when Exception CP15RRTTrap => ec = 0x04; assert from_32;
        when Exception CP14RRTTrap => ec = 0x05; assert from_32;
        when Exception CP14DTTTrap => ec = 0x06; assert from_32;
        when Exception AdvSIMDFPAccessTrap => ec = 0x07;
        when Exception FPIDTrap => ec = 0x08;
        when Exception PACTrap => ec = 0x09;
        when Exception OtherTrap => ec = 0x0A;
        when Exception GPC => ec = 0x1E;
        when Exception CP14RRTTrap => ec = 0x0C; assert from_32;
        when Exception BranchTarget => ec = 0x0D;
        when Exception IllegalState => ec = 0x0E; il_is_valid = FALSE;
        when Exception SupervisorCall => ec = if from_32 then 0x11 else 0x15;
        when Exception HypervisorCall => ec = if from_32 then 0x12 else 0x16;
        when Exception MonitorCall => ec = if from_32 then 0x13 else 0x17;
        when Exception SystemRegisterTrap => ec = 0x18; assert !from_32;
        when Exception SystemRegister128Trap => ec = 0x14; assert !from_32;
        when Exception SVEAccessTrap => ec = 0x19; assert !from_32;
        when Exception ERetTrap => ec = 0x1A; assert !from_32;
        when Exception PACFail => ec = 0x1C; assert !from_32;
        when Exception SMEAccessTrap => ec = 0x1D; assert !from_32;
        when Exception InstructionAbort =>
            ec = if target_el == PSTATE.EL then 0x21 else 0x20;
            il_is_valid = FALSE;
        when Exception PCAlignment => ec = 0x22; il_is_valid = FALSE;
        when Exception DataAbort => ec = if target_el == PSTATE.EL then 0x25 else 0x24;
        when Exception NV2DataAbort => ec = 0x25;
        when Exception SPAlignment => ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception MemCpyMemSet => ec = 0x27;
        when Exception GCSFail => ec = 0x2D; assert !from_32;
        when Exception FPTrappedException => ec = if from_32 then 0x28 else 0x2C;
        when Exception SError => ec = 0x2F; il_is_valid = FALSE;
        when Exception Breakpoint =>
            ec = if target_el == PSTATE.EL then 0x31 else 0x30;
            il_is_valid = FALSE;
        when Exception SoftwareStep =>
            ec = if target_el == PSTATE.EL then 0x33 else 0x32;
            il_is_valid = FALSE;
        when Exception Watchpoint =>
            ec = if target_el == PSTATE.EL then 0x35 else 0x34;
            il_is_valid = FALSE;
        when Exception NV2Watchpoint => ec = 0x35; il_is_valid = FALSE;
        when Exception SoftwareBreakpoint => ec = if from_32 then 0x38 else 0x3C;
        when Exception VectorCatch => ec = 0x3A; il_is_valid = FALSE; assert from_32;
        when Exception Profiling => ec = 0x3D;
        otherwise => unreachable;
    end;

    var il : bit;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    end;
    assert from_32 || il == '1'; // AArch64 instructions always 32-bit

    return (ec,il);
end;
```


Library pseudocode for aarch64/exceptions/exceptions/AArch64_ReportException

```
// AArch64_ReportException()
// =====
// Report syndrome information for exception taken to AArch64 state.

func AArch64_ReportException(except : ExceptionRecord, target_el : bits(2))
begin

    let exceptype : Exception = except.exceptype;

    var (ec,il) = AArch64\_ExceptionClass(exceptype, target_el);
    let iss : bits(25) = except.syndrome.iss;
    let iss2 : bits(24) = except.syndrome.iss2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss[24] == '0' then
        il = '1';
    end;

    ESR\_EL(target_el) = (Zeros{8} :: // [63:56]
                        iss2      :: // [55:32]
                        ec[5:0]   :: // [31:26]
                        il        :: // [25]
                        iss);      // [24:0]

    if exceptype IN {
        Exception\_InstructionAbort,
        Exception\_PCAlignment,
        Exception\_DataAbort,
        Exception\_NV2DataAbort,
        Exception\_NV2Watchpoint,
        Exception\_GPC,
        Exception\_Watchpoint
    } then
        FAR\_EL(target_el) = except.vaddress;
    else
        FAR\_EL(target_el) = ARBITRARY : bits(64);
    end;

    if except.ipavalid then
        HPFAR_EL2() [NUM_PABITS-9:4] = except.ipaddress[NUM\_PABITS-1:12];
        if IsSecureEL2Enabled() && CurrentSecurityState() == SS\_Secure then
            HPFAR_EL2().NS = except.NS;
        else
            HPFAR_EL2().NS = '0';
        end;
    elseif target_el == EL2 then
        HPFAR_EL2() [NUM\_PABITS-9:4] = ARBITRARY : bits(NUM\_PABITS-12);
    end;

    if except.pavalid then
        var faultaddr : bits(64) = ZeroExtend{}(except.paddress.address);
        if IsFeatureImplemented(FEAT_RME) then
            case except.paddress.paspace of
                when PAS\_Secure => faultaddr[63:62] = '00';
                when PAS\_NonSecure => faultaddr[63:62] = '10';
                when PAS\_Root => faultaddr[63:62] = '01';
                when PAS\_Realm => faultaddr[63:62] = '11';
            end;
            if exceptype == Exception\_GPC then
                faultaddr[11:0] = Zeros{12};
            end;
        else
            faultaddr[63] = if except.paddress.paspace == PAS\_NonSecure then '1' else '0';
        end;
        PFAR\_EL(target_el) = faultaddr;
    elseif (IsFeatureImplemented(FEAT_PFAR) ||
            (IsFeatureImplemented(FEAT_RME) && target_el == EL3)) then
        PFAR\_EL(target_el) = ARBITRARY : bits(64);
    end;
    return;
end;
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64_ResetControlRegisters

```
// AArch64_ResetControlRegisters()
// =====
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

impdef func AArch64_ResetControlRegisters(cold_reset : boolean)
begin
    return;
end;
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64_TakeReset

```
// AArch64_TakeReset()
// =====
// Reset into AArch64 state

func AArch64_TakeReset(cold_reset : boolean)
begin
    assert HaveAArch64();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;
    end;

    // Reset System registers and other system components
    AArch64_ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1'; // Select stack pointer
    PSTATE.[D,A,I,F] = '1111'; // All asynchronous exceptions masked
    PSTATE.SS = '0'; // Clear software step bit
    PSTATE.DIT = '0'; // PSTATE.DIT is reset to 0 when resetting into AArch64
    if IsFeatureImplemented(FEAT_PAuth_LR) then
        PSTATE.PACM = '0'; // PAC modifier
    end;
    if IsFeatureImplemented(FEAT_SME) then
        PSTATE.[SM,ZA] = '00'; // Disable Streaming SVE mode & ZA storage
        ResetSMEState('0');
    end;
    if IsFeatureImplemented(FEAT_SSBS) then
        PSTATE.SSBS = ImpDefBit("PSTATE.SSBS bit at reset");
    end;
    if IsFeatureImplemented(FEAT_GCS) then
        PSTATE.EXLOCK = '0'; // PSTATE.EXLOCK is reset to 0 when resetting into AArch64
    end;
    if IsFeatureImplemented(FEAT_UINJ) then
        PSTATE.UINJ = '0'; // PSTATE.UINJ is reset to 0 when resetting into AArch64
    end;
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64_ResetGeneralRegisters();
    if IsFeatureImplemented(FEAT_SME) || IsFeatureImplemented(FEAT_SVE) then
        ResetSVERegisters();
    else
        AArch64_ResetSIMDFPRegisters();
    end;
    AArch64_ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    var rv : bits(64); // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3();
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2();
    else
        rv = RVBAR_EL1();
    end;

    // The reset vector must be correctly aligned
    let pamax : AddressSize = AArch64_PAMax();
    assert IsZero(rv[63:pamax]) && IsZero(rv[1:0]);

    let branch_conditional : boolean = FALSE;
    EDPRSR().R = '0'; // Leaving Reset State.
    BranchTo{64}(rv, BranchType RESET, branch_conditional);
end;
```

Library pseudocode for aarch64/exceptions/exceptions/GetESR_ELx_DFSC

```
// GetESR_ELx_DFSC()
// =====
// Query the ESR_ELx to get the DFSC field.

func GetESR_ELx_DFSC(el : bits(2)) => bits(6)
begin
    let ec : bits(6) = ESR\_EL(el).EC;
    assert ec IN {'10010x', '101111', '11010x'};

    return ESR\_EL(el)[5:0];
end;
```

Library pseudocode for aarch64/exceptions/exceptions/GetESR_ELx_ExType

```
// GetESR_ELx_ExType()
// =====
// Query the ESR_ELx to get the ExType field.

func GetESR_ELx_ExType(el : bits(2)) => bits(4)
begin
    let ec : bits(6) = ESR\_EL(el).EC;
    assert ec == '101101';

    return ESR\_EL(el)[23:20];
end;
```

Library pseudocode for aarch64/exceptions/exceptions/GetESR_ELx_IFSC

```
// GetESR_ELx_IFSC()
// =====
// Query the ESR_ELx to get the IFSC field.

func GetESR_ELx_IFSC(el : bits(2)) => bits(6)
begin
    let ec : bits(6) = ESR\_EL(el).EC;
    assert ec IN {'1100xx', '10000x'};

    return ESR\_EL(el)[5:0];
end;
```

Library pseudocode for aarch64/exceptions/ieeefp/AArch64_FPTrappedException

```
// AArch64_FPTrappedException()
// =====

func AArch64_FPTrappedException(is_ase : boolean, accumulated_exceptions : bits(8))
begin
    var except : ExceptionRecord = ExceptionSyndrome(Exception\_FPTrappedException);
    if is_ase then
        if ImpDefBool("vector instructions set TFV to 1") then
            except.syndrome.iss[23] = '1'; // TFV
        else
            except.syndrome.iss[23] = '0'; // TFV
        end;
    else
        except.syndrome.iss[23] = '1'; // TFV
    end;
    except.syndrome.iss[10:8] = ARBITRARY : bits(3); // VECITR
    if except.syndrome.iss[23] == '1' then
        except.syndrome.iss[7,4:0] = accumulated_exceptions[7,4:0]; // IDF, IXF, UFF, OFF, DZF, IOF
    else
        except.syndrome.iss[7,4:0] = ARBITRARY : bits(6);
    end;

    let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR\_EL2().TGE == '1';

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64_CallHypervisor

```
// AArch64_CallHypervisor()
// =====
// Performs a HVC call

func AArch64_CallHypervisor(immediate : bits(16))
begin
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32_ITAdvance(); end;
    SSAdvance();
    let preferred_exception_return : bits(64) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome.iss[15:0] = immediate;

    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    if PSTATE.EL == EL3 then
        AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64_CallSecureMonitor

```
// AArch64_CallSecureMonitor()
// =====

func AArch64_CallSecureMonitor(immediate : bits(16))
begin
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32_ITAdvance(); end;
    HSAdvance();
    SSAdvance();
    let preferred_exception_return : bits(64) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_MonitorCall);
    except.syndrome.iss[15:0] = immediate;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64_CallSupervisor

```
// AArch64_CallSupervisor()
// =====
// Calls the Supervisor

func AArch64_CallSupervisor(immediate : bits(16))
begin
    if UsingAArch32() then AArch32_ITAdvance(); end;
    SSAdvance();
    let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1';

    let preferred_exception_return : bits(64) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_SupervisorCall);
    except.syndrome.iss[15:0] = immediate;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elseif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```



```

// AArch64_TakeException()
// =====
// Take an exception to an Exception level using AArch64.

noreturn func AArch64_TakeException(target_el : bits(2), exception_in : ExceptionRecord,
                                   preferred_exception_return : bits(64), vect_offset_in : integer)
begin
  assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
  if Halted() then
    AArch64_TakeExceptionInDebugState(target_el, exception_in);
  end;
  var except : ExceptionRecord = exception_in;
  var sync_errors : boolean;
  var iesb_req : boolean;
  if IsFeatureImplemented(FEAT_IESB) then
    sync_errors = SCTLR\_EL(target_el).IESB == '1';
    if IsFeatureImplemented(FEAT_DoubleFault) then
      sync_errors = sync_errors || (SCR_EL3().[EA,NMEA] == '11' && target_el == EL3);
    end;
  end;
  if sync_errors && InsertIESBBeforeException(target_el) then
    SynchronizeErrors();
    if except.exceptype != Exception\_SError then
      iesb_req = FALSE;
      sync_errors = FALSE;
      TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    end;
  end;
  else
    sync_errors = FALSE;
  end;

  var brbe_source_allowed : boolean = FALSE;
  var brbe_source_address : bits(64) = Zeros{};
  if IsFeatureImplemented(FEAT_BRBE) then
    brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
    brbe_source_address = preferred_exception_return;
  end;

  if !IsFeatureImplemented(FEAT_ExS) || SCTLR\_EL(target_el).EIS == '1' then
    // Synchronize the context, including Instruction Fetch Barrier effect
    SynchronizeContext();
  elseif !(except.exceptype == Exception\_SoftwareBreakpoint ||
           (except.exceptype IN {Exception\_SupervisorCall,
                                Exception\_HypervisorCall,
                                Exception\_MonitorCall} &&
            !except.trappedsyscallinst)) then
    InstructionFetchBarrier();
  end;

  // If coming from AArch32 state, the top parts of the X() registers might be set to zero
  let from_32 : boolean = UsingAArch32();
  if from_32 then AArch64\_MaybeZeroRegisterUppers(); end;
  if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
    ResetSVEState();
  else
    MaybeZeroSVEUppers(target_el);
  end;

  var vect_offset : integer = vect_offset_in;
  if UInt(target_el) > UInt(PSTATE.EL) then
    var lower_32 : boolean;
    if target_el == EL3 then
      if EL2Enabled() then
        lower_32 = ELUsingAArch32(EL2);
      else
        lower_32 = ELUsingAArch32(EL1);
      end;
    elseif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
      lower_32 = ELUsingAArch32(EL0);
    else
      lower_32 = ELUsingAArch32(target_el - 1);
    end;
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);
  end;

  elseif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;
  end;

  assert vect_offset < 2048;
  let vect_base : bits(64) = VBAR\_EL(target_el)[63:11]::Zeros{11};
  let target_vector : bits(64) = vect_base + vect_offset;

  var spsr : bits(64) = GetPSRFFromPSTATE{}(AArch64\_NonDebugState);

  if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
    if EffectiveHCR\_EL2\_NVx() IN {'x01', '111'} then
      spsr[3:2] = '10';
    end;
  end;
end;

```

```

if IsFeatureImplemented(FEAT_BTI) && !UsingAArch32() then
    var zero_btype : boolean;
    // SPSR_ELx[].BTYPE is only guaranteed valid for these exception types
    if except.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
                           Exception_SoftwareStep, Exception_PCAAlignment,
                           Exception_InstructionAbort, Exception_Breakpoint,
                           Exception_VectorCatch, Exception_SoftwareBreakpoint,
                           Exception_IllegalState, Exception_BranchTarget} then
        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
    end;
    if zero_btype then spsr[11:10] = '00'; end;
end;

if (IsFeatureImplemented(FEAT_NV2) &&
    except.exceptype == Exception_NV2DataAbort && target_el == EL3) then
    // External aborts are configured to be taken to EL3
    except.exceptype = Exception_DataAbort;
end;
if ! except.exceptype IN {Exception_IRQ, Exception_FIQ} then
    AArch64_ReportException(except, target_el);
end;

if IsFeatureImplemented(FEAT_BRBE) then
    BRBEEException(except, brbe_source_allowed, brbe_source_address,
                   target_vector, target_el, except.trappedsyscallinst);
end;

if IsFeatureImplemented(FEAT_GCS) then
    if PSTATE.EL == target_el then
        if GetCurrentEXLOCKEN() then
            PSTATE.EXLOCK = '1';
        else
            PSTATE.EXLOCK = '0';
        end;
    else
        PSTATE.EXLOCK = '0';
    end;
end;

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR_ELx() = spsr;
ELR_ELx() = preferred_exception_return;
PSTATE.SS = '0';
if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR_ELx().DSSBS; end;
if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = '1'; end;
if IsFeatureImplemented(FEAT_NMI) then
    PSTATE.ALLINT = NOT SCTLR_ELx().SPINTMASK;
end;
PSTATE.[D,A,I,F] = '1111';
if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1'; end;
PSTATE.IL = '0';
if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;
if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
if IsFeatureImplemented(FEAT_PAN) && SCTLR_ELx().SPAN == '0' then
    if PSTATE.EL == EL1 then
        PSTATE.PAN = '1';
    elsif PSTATE.EL == EL2 && ELIsInHost(EL0) then
        PSTATE.PAN = '1';
    elsif (PSTATE.EL == EL2 && ELIsInHost(EL2) &&
        ImpDefBool("Set PAN as 1 when E2H,TGE is '10'")) then
        PSTATE.PAN = '1';
    end;
end;

if from_32 then
    PSTATE.IT = '00000000';
    PSTATE.T = '0';
end;
if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00'; end;
let branch_conditional : boolean = FALSE;
BranchTo{64}(target_vector, BranchType_EXCEPTION, branch_conditional);
CheckExceptionCatch(TRUE);

if sync_errors then
    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
end;

EndOfInstruction(EndOfInstructionReason_Exception);
end;

```

Library pseudocode for aarch64/exceptions/traps/AArch64_AAarch32SystemAccessTrap

```
// AArch64_AAarch32SystemAccessTrap()
// =====
// Trapped AArch32 System register access.

func AArch64_AAarch32SystemAccessTrap(target_el : bits(2), ec : integer)
begin
    assert HaveEL(target_el) && target_el != ELO && UInt(target_el) >= UInt(PSTATE.EL);

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    let except : ExceptionRecord = AArch64_AAarch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```


Library pseudocode for aarch64/exceptions/traps/AArch64_AArch32SystemAccessTrapSyndrome

```
// AArch64_AArch32SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

func AArch64_AArch32SystemAccessTrapSyndrome(instr : bits(32), ec : integer) => ExceptionRecord
begin
    var except : ExceptionRecord;

    case ec of
        when 0x0 => except = ExceptionSyndrome(Exception Uncategorized);
        when 0x3 => except = ExceptionSyndrome(Exception CP15RTTrap);
        when 0x4 => except = ExceptionSyndrome(Exception CP15RRTTrap);
        when 0x5 => except = ExceptionSyndrome(Exception CP14RTTrap);
        when 0x6 => except = ExceptionSyndrome(Exception CP14DTTrap);
        when 0x7 => except = ExceptionSyndrome(Exception AdvSIMDFPAccessTrap);
        when 0x8 => except = ExceptionSyndrome(Exception FPIDTrap);
        when 0xC => except = ExceptionSyndrome(Exception CP14RRTTrap);
        otherwise => unreachable;
    end;

    var iss : bits(20) = Zeros{};

    if except.exceptype == Exception Uncategorized then
        return except;
    elsif except.exceptype IN {Exception FPIDTrap, Exception CP14RTTrap,
                             Exception CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if except.exceptype != Exception FPIDTrap then // When trap is not for VMRS
            iss[19:17] = instr[7:5]; // opc2
            iss[16:14] = instr[23:21]; // opc1
            iss[13:10] = instr[19:16]; // CRn
            iss[4:1] = instr[3:0]; // CRm
        else
            iss[19:17] = '000';
            iss[16:14] = '111';
            iss[13:10] = instr[19:16]; // reg
            iss[4:1] = '0000';
        end;

        if instr[20] == '1' && instr[15:12] == '1111' then // MRC, Rt==15
            iss[9:5] = '1111';
        elsif instr[20] == '0' && instr[15:12] == '1111' then // MCR, Rt==15
            iss[9:5] = ARBITRARY : bits(5);
        else
            iss[9:5] = LookUpRIndex(UInt(instr[15:12]), PSTATE.M)[4:0];
        end;
    elsif except.exceptype IN {Exception CP14RRTTrap, Exception AdvSIMDFPAccessTrap,
                             Exception CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss[19:16] = instr[7:4]; // opc1
        if instr[19:16] == '1111' then // Rt2==15
            iss[14:10] = ARBITRARY : bits(5);
        else
            iss[14:10] = LookUpRIndex(UInt(instr[19:16]), PSTATE.M)[4:0];
        end;

        if instr[15:12] == '1111' then // Rt==15
            iss[9:5] = ARBITRARY : bits(5);
        else
            iss[9:5] = LookUpRIndex(UInt(instr[15:12]), PSTATE.M)[4:0];
        end;
        iss[4:1] = instr[3:0]; // CRm
    elsif except.exceptype == Exception CP14DTTrap then
        // Trapped LDC/STC
        iss[19:12] = instr[7:0]; // imm8
        iss[4] = instr[23]; // U
        iss[2:1] = instr[24,21]; // P,W
        if instr[19:16] == '1111' then // Rn==15, LDC(Literal addressing)/STC
            iss[9:5] = ARBITRARY : bits(5);
            iss[3] = '1';
        end;
    end;

    iss[0] = instr[20]; // Direction

    except.syndrome.iss[24:20] = ConditionSyndrome();
    except.syndrome.iss[19:0] = iss;

    return except;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_AdvSIMDFPAccessTrap

```
// AArch64_AdvSIMDFPAccessTrap()
// =====
// Trapped access to Advanced SIMD or FP registers due to CPACR.

func AArch64_AdvSIMDFPAccessTrap(target_el : bits(2))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var except : ExceptionRecord;
    let vect_offset : integer = 0x0;

    let route_to_el2 : boolean = (target_el == EL1 && EL2Enabled() && HCR_EL2().TGE == '1');

    if route_to_el2 then
        except = ExceptionSyndrome(Exception Uncategorized);
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        except = ExceptionSyndrome(Exception AdvSIMDFPAccessTrap);
        except.syndrome.iss[24:20] = ConditionSyndrome();
        AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
    end;

    return;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_CheckCP15InstrCoarseTraps

```
// AArch64_CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained AArch32 traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTLR_ELx.

func AArch64_CheckCP15InstrCoarseTraps(CRn : integer, nreg : integer, CRm : integer)
begin
    let trapped_encoding : boolean = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
                                       (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
                                       (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCTLR_EL1.TIDCP.
    if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && !IsInHost() &&
        !ELUsingAArch32(EL1) && SCTLR_EL1().TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
            AArch64 AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64 AArch32SystemAccessTrap(EL1, 0x3);
        end;
    end;

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTLR_EL2.TIDCP.
        if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && IsInHost() &&
            SCTLR_EL2().TIDCP == '1' && trapped_encoding) then
            AArch64 AArch32SystemAccessTrap(EL2, 0x3);
        end;

        let major : integer = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2().CRn/HSTR().CRm
        // and MRC and MCR disabled by HCR_EL2.TIDCP.
        if ((!IsInHost() && !major IN {4,14} && HSTR_EL2()[major] == '1') ||
            (HCR_EL2().TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                ImpDefBool("UNDEF unallocated CP15 access at EL0")) then
                Undefined();
            end;
            AArch64 AArch32SystemAccessTrap(EL2, 0x3);
        end;
    end;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_CheckFPAdvSIMDEnabled

```
// AArch64_CheckFPAdvSIMDEnabled()
// =====

func AArch64_CheckFPAdvSIMDEnabled()
begin
    AArch64 CheckFPEEnabled();
    // Check for illegal use of Advanced
    // SIMD in Streaming SVE Mode
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType Streaming, PSTATE.EL);
    end;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_CheckFPAdvSIMDTrap

```
// AArch64_CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

func AArch64_CheckFPAdvSIMDTrap()
begin
    if HaveEL(EL3) && CPTR_EL3().TFP == '1' && EL3SDDUndefPriority() then
        Undefined();
    end;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if ELIsInHost(EL2) then
            var disabled : boolean;
            case CPTR_EL2().FPEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        else
            if CPTR_EL2().TFP == '1' then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        end;
    end;

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3().TFP == '1' then
            if EL3SDDUndef() then
                Undefined();
            else
                AArch64_AdvSIMDFPAccessTrap(EL3);
            end;
        end;
    end;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_CheckFPEnabled

```
// AArch64_CheckFPEnabled()
// =====
// Check against CPACR

func AArch64_CheckFPEnabled()
begin
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        var disabled : boolean;
        case CPACR_EL1().FPEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then AArch64_AdvSIMDFPAccessTrap(EL1); end;
    end;

    AArch64_CheckFPAdvSIMDTrap(); // Also check against CPTR_EL2 and CPTR_EL3
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_CheckForERetTrap

```
// AArch64_CheckForERetTrap()
// =====
// Check for trap on ERET, ERETA, ERETB instruction

func AArch64_CheckForERetTrap(eret_with_pac : boolean, pac_uses_key_a : boolean)
begin
    if PSTATE.EL == EL1 && EL2Enabled() then
        var route_to_el2 : boolean = FALSE;

        // Check for a fine-grained trap by the hypervisor.
        if (IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR\_EL3\(\).FGTE == '1') &&
            HFGITR\_EL2\(\).ERET == '1') then
            route_to_el2 = TRUE;
        // Check for a trap by the Effective value of the HCR_EL2.NV bit
        elsif EffectiveHCR\_EL2\_NVx() [0] == '1' then
            route_to_el2 = TRUE;
        end;

    if route_to_el2 then
        let preferred_exception_return : bits(64) = ThisInstrAddr{}();
        let vect_offset : integer = 0x0;
        var except : ExceptionRecord = ExceptionSyndrome(Exception\_ERetTrap);

        if !eret_with_pac then // ERET
            except.syndrome.iss[1] = '0';
            except.syndrome.iss[0] = '0'; // RES0
        else
            except.syndrome.iss[1] = '1';
            except.syndrome.iss[0] = if pac_uses_key_a then '0' else '1';
        end;

        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    end;
end;

return;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_CheckForSMCUnDefOrTrap

```
// AArch64_CheckForSMCUnDefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

func AArch64_CheckForSMCUnDefOrTrap(imm : bits(16))
begin
    if PSTATE.EL == EL0 then Undefined(); end;
    if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR\_EL2\(\).TSC == '1') &&
        HaveEL(EL3) && SCR\_EL3\(\).SMD == '1') then
        Undefined();
    end;
    var route_to_el2 : boolean = FALSE;
    if !HaveEL(EL3) then
        if (PSTATE.EL == EL1 && EL2Enabled() && HCR\_EL2\(\).TSC == '1' &&
            (EffectiveHCR\_EL2\_NVx() == 'xx1' ||
            (ImpDefBool("Trap SMC execution at EL1 to EL2")))) then
            route_to_el2 = TRUE;
        else
            Undefined();
        end;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR\_EL2\(\).TSC == '1';
    end;
    if route_to_el2 then
        let preferred_exception_return : bits(64) = ThisInstrAddr{}();
        let vect_offset : integer = 0x0;
        var except : ExceptionRecord = ExceptionSyndrome(Exception\_MonitorCall);
        except.syndrome.iss[15:0] = imm;
        except.trappedsyscallinst = TRUE;
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_CheckForSVCTrap

```
// AArch64_CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

func AArch64_CheckForSVCTrap(immediate : bits(16))
begin
    if IsFeatureImplemented(FEAT_FGT) then
        var route_to_el2 : boolean = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!UsingAArch32() && !ELUsingAArch32(EL1) &&
                           EL2Enabled() && HFGITR_EL2().SVC_EL0 == '1' &&
                           (!IsInHost() && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1')));

        elsif PSTATE.EL == EL1 then
            route_to_el2 = (EL2Enabled() && HFGITR_EL2().SVC_EL1 == '1' &&
                           (!HaveEL(EL3) || SCR_EL3().FGTEn == '1'));

        end;

        if route_to_el2 then
            var except : ExceptionRecord = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome.iss[15:0] = immediate;
            except.trappedsyscallinst = TRUE;
            let preferred_exception_return : bits(64) = ThisInstrAddr();
            let vect_offset : integer = 0x0;

            AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);

        end;
    end;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_CheckForWFXTrap

```
// AArch64_CheckForWFXTrap()
// =====
// Checks for a trap on a WFE, WFET, WFI or WFIT instruction.

func AArch64_CheckForWFXTrap(wfxtype : WFXType) => (boolean, bits(2))
begin
    let is_wfe : boolean = wfxtype IN {WFXType_WFE, WFXType_WFET};
    var target_el : bits(2);
    var trap : boolean = FALSE;

    if HaveEL(EL3) && EL3SDDUndefPriority() && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        // If the trap is enabled, the instruction will be UNDEFINED because EDSCR.SDD is 1.
        trap = (if is_wfe then SCR_EL3().TWE else SCR_EL3().TWI) == '1';
        target_el = EL3;
    end;

    if !trap && PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        trap = (if is_wfe then SCTLRL_ELx().nTWE else SCTLRL_ELx().nTWI) == '0';
        target_el = EL1;
    end;

    if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        trap = (if is_wfe then HCR_EL2().TWE else HCR_EL2().TWI) == '1';
        target_el = EL2;
    end;

    if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        trap = (if is_wfe then SCR_EL3().TWE else SCR_EL3().TWI) == '1';
        target_el = EL3;
    end;
    return (trap, target_el);
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_CheckIllegalState

```
// AArch64_CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

func AArch64_CheckIllegalState()
begin
    if PSTATE.IL == '1' then
        let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1';

        let preferred_exception_return : bits(64) = ThisInstrAddr{}();
        let vect_offset : integer = 0x0;

        let except : ExceptionRecord = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
        else
            AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
        end;
    end;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_MonitorModeTrap

```
// AArch64_MonitorModeTrap()
// =====
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

func AArch64_MonitorModeTrap()
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    let except : ExceptionRecord = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    end;
    AArch64_TakeException(EL3, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_OtherInstrTrap

```
// AArch64_OtherInstrTrap()
// =====
// Trapped access to the following instructions:
// - LD64B
// - ST64B
// - ST64BV
// - ST64BV0
// - PSB
// - TSB

func AArch64_OtherInstrTrap(target_el : bits(2), iss : bits(25))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_OtherTrap);
    except.syndrome.iss = iss;
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_SystemAccessTrap

```
// AArch64_SystemAccessTrap()
// =====
// Trapped access to AArch64 System register or system instruction.

func AArch64_SystemAccessTrap(target_el : bits(2), ec : integer)
begin
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    let except : ExceptionRecord = AArch64_SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_SystemAccessTrapSyndrome

```
// AArch64_SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

func AArch64_SystemAccessTrapSyndrome(instr_in : bits(32), ec : integer) => ExceptionRecord
begin
    var except : ExceptionRecord;
    var instr : bits(32) = instr_in;
    case ec of
        when 0x0 => // Trapped access due to unknown reason.
            except = ExceptionSyndrome(Exception Uncategorized);
        when 0x7 => // Trapped access to SVE, Advance SIMD&FP System register.
            except = ExceptionSyndrome(Exception AdvSIMDFPAccessTrap);
            except.syndrome.iss[24:20] = ConditionSyndrome();
        when 0x14 => // Trapped access to 128-bit System register or
            // 128-bit System instruction.
            except = ExceptionSyndrome(Exception SystemRegister128Trap);
            instr = ThisInstr();
            except.syndrome.iss[21:20] = instr[20:19]; // Op0
            except.syndrome.iss[19:17] = instr[7:5]; // Op2
            except.syndrome.iss[16:14] = instr[18:16]; // Op1
            except.syndrome.iss[13:10] = instr[15:12]; // CRn
            except.syndrome.iss[9:6] = instr[4:1]; // Rt
            except.syndrome.iss[4:1] = instr[11:8]; // CRm
            except.syndrome.iss[0] = instr[21]; // Direction
        when 0x18 => // Trapped access to System register or system instruction.
            except = ExceptionSyndrome(Exception SystemRegisterTrap);
            instr = ThisInstr();
            except.syndrome.iss[21:20] = instr[20:19]; // Op0
            except.syndrome.iss[19:17] = instr[7:5]; // Op2
            except.syndrome.iss[16:14] = instr[18:16]; // Op1
            except.syndrome.iss[13:10] = instr[15:12]; // CRn
            except.syndrome.iss[9:5] = instr[4:0]; // Rt
            except.syndrome.iss[4:1] = instr[11:8]; // CRm
            except.syndrome.iss[0] = instr[21]; // Direction
        when 0x19 => // Trapped access to SVE System register
            except = ExceptionSyndrome(Exception SVEAccessTrap);
        when 0x1D => // Trapped access to SME System register
            except = ExceptionSyndrome(Exception SMEAccessTrap);
        otherwise =>
            unreachable;
    end;

    return except;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_Undefined

```
// AArch64_Undefined()
// =====

noreturn func AArch64_Undefined()
begin

    let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR\_EL2().TGE == '1';
    let preferred_exception_return : bits(64) = ThisInstrAddr();
    let vect_offset : integer = 0x0;

    let except : ExceptionRecord = ExceptionSyndrome(Exception Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/exceptions/traps/AArch64_WFxFTrap

```
// AArch64_WFxFTrap()
// =====
// Generate an exception for a trapped WFE, WFI, WFET or WFIT instruction.

func AArch64_WFxFTrap(wfxtype : WFxType, target_el : bits(2))
begin
    assert UInt(target_el) > UInt(PSTATE.EL);
    let preferred_exception_return : bits(64) = ThisInstrAddr();
    let vect_offset : integer = 0x0;
    var except : ExceptionRecord = ExceptionSyndrome(Exception_WFxFTrap);
    except.syndrome.iss[24:20] = ConditionSyndrome();
    case wfxtype of
        when WFxType_WFI =>
            except.syndrome.iss[1:0] = '00';
        when WFxType_WFE =>
            except.syndrome.iss[1:0] = '01';
        when WFxType_WFIT =>
            except.syndrome.iss[1:0] = '10';
            except.syndrome.iss[2] = '1'; // Register field is valid
            except.syndrome.iss[9:5] = ThisInstr()[4:0];
        when WFxType_WFET =>
            except.syndrome.iss[1:0] = '11';
            except.syndrome.iss[2] = '1'; // Register field is valid
            except.syndrome.iss[9:5] = ThisInstr()[4:0];
    end;

    if target_el == EL1 && EL2Enabled() && HCR_EL2().TGE == '1' then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabled

```
// CheckLDST64BEnabled()
// =====
// Checks for trap on ST64B and LD64B instructions

func CheckLDST64BEnabled()
begin
    var trap : boolean = FALSE;
    let iss : bits(25) = 0x2[24:0];
    var target_el : bits(2);

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTL_EL1().EnALS == '0';
            target_el = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        else
            trap = SCTL_EL2().EnALS == '0';
            target_el = EL2;
        end;
    else
        target_el = EL1;
    end;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2().EnALS == '0';
        target_el = EL2;
    end;

    if trap then AArch64_OtherInstrTrap(target_el, iss); end;
end;
```


Library pseudocode for aarch64/exceptions/traps/CheckST64BV0Enabled

```
// CheckST64BV0Enabled()
// =====
// Checks for trap on ST64BV0 instruction

func CheckST64BV0Enabled()
begin
    var trap : boolean = FALSE;
    let iss : bits(25) = 0x1[24:0];
    var target_el : bits(2);

    if (PSTATE.EL != EL3 && HaveEL(EL3) &&
        SCR_EL3().EnAS0 == '0' && EL3SDDUndefPriority()) then
        Undefined();
    end;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTL_EL1().EnAS0 == '0';
            target_el = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        else
            trap = SCTL_EL2().EnAS0 == '0';
            target_el = EL2;
        end;
    end;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2().EnAS0 == '0';
        target_el = EL2;
    end;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3().EnAS0 == '0';
        target_el = EL3;
    end;

    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            Undefined();
        else
            AArch64_OtherInstrTrap(target_el, iss);
        end;
    end;
end;
```

Library pseudocode for aarch64/exceptions/traps/CheckST64BVEnabled

```
// CheckST64BVEnabled()
// =====
// Checks for trap on ST64BV instruction

func CheckST64BVEnabled()
begin
    var trap : boolean = FALSE;
    let iss : bits(25) = 0x0[24:0];
    var target_el : bits(2);

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTL_EL1().EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        else
            trap = SCTL_EL2().EnASR == '0';
            target_el = EL2;
        end;
    end;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2().EnASR == '0';
        target_el = EL2;
    end;

    if trap then AArch64_OtherInstrTrap(target_el, iss); end;
end;
```

Library pseudocode for aarch64/exceptions/traps/WFETrapDelay

```
// WFETrapDelay()
// =====
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

func WFETrapDelay(target_el : bits(2)) => (boolean, integer)
begin
    var delay_enabled : boolean;
    var delay : integer;
    case target_el of
        when EL1 =>
            if !IsInHost() then
                delay_enabled = SCTLR_EL1().TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL1().TWEDEL) + 8);
            else
                delay_enabled = SCTLR_EL2().TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL2().TWEDEL) + 8);
            end;
        when EL2 =>
            assert EL2Enabled();
            delay_enabled = HCR_EL2().TWEDEn == '1';
            delay         = 1 << (UInt(HCR_EL2().TWEDEL) + 8);
        when EL3 =>
            delay_enabled = SCR_EL3().TWEDEn == '1';
            delay         = 1 << (UInt(SCR_EL3().TWEDEL) + 8);
    end;
    return (delay_enabled, delay);
end;
```

Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay

```
// WaitForEventUntilDelay()
// =====
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires, including potentially
// some IMPLEMENTATION SPECIFIC additional delay, FALSE otherwise.

impdef func WaitForEventUntilDelay(delay_enabled : boolean, delay : integer) => boolean
begin
    return FALSE;
end;
```



```

// AArch64_FaultSyndrome()
// =====
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to an Exception level using AArch64.

func AArch64_FaultSyndrome(exceptiontype : Exception, fault : FaultRecord,
                           target_el : bits(2)) => IssType
begin
    assert fault.statuscode != Fault_None;

    var isstype : IssType;
    isstype.iss = Zeros(25);
    isstype.iss2 = Zeros(24);

    let d_side : boolean = exceptiontype IN {Exception_DataAbort, Exception_NV2DataAbort,
                                             Exception_Watchpoint, Exception_NV2Watchpoint};
    if IsFeatureImplemented(FEAT_RAS) && fault.statuscode == Fault_SyncExternal then
        let errstate : ErrorState = PErrorState(fault);
        isstype.iss[12:11] = AArch64_EncodeSyncErrorSyndrome(errstate); // SET
    end;

    if d_side then
        if fault.accessdesc.acctype == AccessType_GCS then
            isstype.iss2[8] = '1';
        end;
        if exceptiontype IN {Exception_Watchpoint, Exception_NV2Watchpoint} then
            isstype.iss[23:0] = WatchpointRelatedSyndrome(fault);
        end;

        if AArch64_InstructionSyndromeValid(fault) then
            if fault.accessdesc.ls64 then
                (isstype.iss2, isstype.iss[24:14]) = LS64InstructionSyndrome();
            else
                isstype.iss[24:14] = LSInstructionSyndrome();
            end;
        end;

        if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
            isstype.iss[13] = '1'; // Fault is generated by use of VNCR_EL2
        end;

        if (IsFeatureImplemented(FEAT_LS64) &&
            fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
            isstype.iss[12:11] = GetLoadStoreType();
        end;

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss[8] = '1';
        end;

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss[6] = '1';
        elseif fault.statuscode IN {Fault_UnsupportedAtomicHWUpdate, Fault_Exclusive} then
            isstype.iss[6] = ARBITRARY : bit;
        elseif fault.accessdesc.atomicop && IsExternalAbort(fault) then
            isstype.iss[6] = ARBITRARY : bit;
        else
            isstype.iss[6] = if fault.write then '1' else '0';
        end;
        isstype.iss2[9] = if fault.tagaccess then '1' else '0';
        if fault.statuscode == Fault_Permission then
            isstype.iss2[5] = if fault.dirtybit then '1' else '0';
            isstype.iss2[6] = if fault.overlay then '1' else '0';
            if isstype.iss[24] == '0' then
                isstype.iss[21] = if fault.toplevel then '1' else '0';
            end;
            isstype.iss2[7] = if fault.assuredonly then '1' else '0';
            isstype.iss2[10] = if fault.sltagnotdata then '1' else '0';
        end;

    else
        if (fault.accessdesc.acctype == AccessType_IFETCH &&
            fault.statuscode == Fault_Permission) then
            isstype.iss2[5] = if fault.dirtybit then '1' else '0';
            isstype.iss[21] = if fault.toplevel then '1' else '0';
            isstype.iss2[7] = if fault.assuredonly then '1' else '0';
            isstype.iss2[6] = if fault.overlay then '1' else '0';
        end;
    end;
    if fault.hdbssf then
        isstype.iss2[11] = '1';
    end;

    if (IsFeatureImplemented(FEAT_PFAR) && IsExternalSyncAbort(fault) &&
        !(EL2Enabled() && (HCR_EL2().VM == '1' || HCR_EL2().DC == '1') && target_el == EL1)) then
        isstype.iss[14] = if IsPFARValid(fault) then '1' else '0';
    end;

    if IsExternalAbort(fault) then isstype.iss[9] = fault.extflag; end;

```

```

    isstype.iss[7] = if fault.s2fslwalk then '1' else '0';
    isstype.iss[5:0] = EncodeLDFSC(fault.statuscode, fault.level);

    return isstype;
end;

```

Library pseudocode for aarch64/functions/aborts/AArch64_InstructionSyndromeValid

```

// AArch64_InstructionSyndromeValid()
// =====
// Returns TRUE if ESR_ELx.ISV is '1' for the given Fault.

func AArch64_InstructionSyndromeValid(fault : FaultRecord) => boolean
begin
    if !fault.accessdesc.lowestaddress then
        return FALSE;
    end;

    if fault.accessdesc.ls64 then
        return fault.statuscode IN {Fault AccessFlag, Fault Translation, Fault Permission};
    end;

    if fault.accessdesc.acctype == AccessType GCS then
        return FALSE;
    end;

    if IsSecondStage(fault) && !fault.s2fslwalk then
        return (!IsExternalSyncAbort(fault) ||
            (!IsFeatureImplemented(FEAT_RAS) && IsExternalAbortOnWalk(fault) &&
                ImpDefBool("ISV on second stage translation table walk")));
    end;

    return FALSE;
end;

```

Library pseudocode for aarch64/functions/aborts/EncodeGPCSC

```

// EncodeGPCSC()
// =====
// Function that gives the GPCSC code for types of GPT Fault

func EncodeGPCSC(gpcf : GPCFRecord) => bits(6)
begin
    assert gpcf.level IN {0,1};

    case gpcf.gpcf of
        when GPCF AddressSize => return '00000'::gpcf.level[0];
        when GPCF Walk => return '00010'::gpcf.level[0];
        when GPCF Fail => return '00110'::gpcf.level[0];
        when GPCF EABT => return '01010'::gpcf.level[0];
    end;
end;

```

Library pseudocode for aarch64/functions/aborts/LS64InstructionSyndrome

```

// LS64InstructionSyndrome()
// =====
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field.

impdef func LS64InstructionSyndrome() => (bits(24), bits(11))
begin
    return (Zeros{24}, LSInstructionSyndrome());
end;

```

Library pseudocode for aarch64/functions/aborts/WatchpointFARNotPrecise

```

// WatchpointFARNotPrecise()
// =====
// Returns TRUE If the lowest watchpointed address that is higher than or equal to the address
// recorded in EDWAR might not have been accessed by the instruction, other than the CONSTRAINED
// UNPREDICTABLE condition of watchpoint matching a range of addresses with lowest address 16 bytes
// rounded down and upper address rounded up to nearest 16 byte multiple,
// FALSE otherwise.

impdef func WatchpointFARNotPrecise(fault : FaultRecord) => boolean
begin
    if IsFeatureImplemented(FEAT_SVE) || IsFeatureImplemented(FEAT_SME) then
        return fault.watchptinfo.maybe_false_match;
    end;
    return FALSE;
end;

```

Library pseudocode for aarch64/functions/apas/AArch64_APAS

```
// AArch64_APAS()
// =====
// Decode Xt and perform an APAS operation for the decoded record.

func AArch64_APAS(Xt : bits(64))
begin
    var apas : APASRecord;
    let nse2 : bit = '0';
    apas.paspace = DecodePASpace(nse2, Xt[62], Xt[63]);

    apas.pa = Xt[NUM\_PABITS-1:6] :: '000000';
    apas.target_attributes = Xt[2:0];

    if AArch64\_LocationSupportsAPAS(apas) then
        APAS_OP(apas);
    end;
end;
```

Library pseudocode for aarch64/functions/apas/AArch64_LocationSupportsAPAS

```
// AArch64_LocationSupportsAPAS()
// =====
// Returns TRUE if the given memory location supports the APAS instruction.

impdef func AArch64_LocationSupportsAPAS(apas : APASRecord) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for aarch64/functions/apas/APASRecord

```
// APASRecord
// =====
// Details related to an APAS operation.

type APASRecord of record {
    pa : bits(NUM\_PABITS),
    paspace : PASpace,
    target_attributes : bits(3)
};
```

Library pseudocode for aarch64/functions/apas/APAS_OP

```
// APAS_OP()
// =====
// Sets the PA Space of the address in the APASRecord to the target PA space. If the location
// does not support the APAS instruction or cannot be associated with the indicated PASpace,
// then the instruction has no effect on the location and does not generate an External abort.

impdef func APAS_OP(apas : APASRecord)
begin
    return;
end;
```

Library pseudocode for aarch64/functions/at/AArch64_AT

```
// AArch64_AT()
// =====
// Perform address translation as per AT instructions.

func AArch64_AT(address : bits(64), stage : TranslationStage, el_in : bits(2), ataccess : ATAccess)
begin
    var el : bits(2) = el_in;
    // For stage 1 translation, when HCR_EL2().[E2H, TGE] is {1,1} and requested EL is EL1,
    // the EL2&0 translation regime is used.
    if ELIsInHost(EL0) && el == EL1 && stage == TranslationStage\_1 then
        el = EL2;
    end;

    let ss : SecurityState = SecurityStateAtEL(el);

    let accdesc : AccessDescriptor = CreateAccDescAT(ss, el, ataccess);
    let aligned : boolean = TRUE;

    var fault : FaultRecord = NoFault(accdesc, address);
    let regime : Regime = TranslationRegime(el);

    var addrdesc : AddressDescriptor;
    if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
        if regime == Regime\_EL2 || TTBCR().EAE == '1' then
            (fault, addrdesc) = AArch32\_S1TranslateLD(fault, regime, address[31:0], aligned,
                accdesc);
        else
            (fault, addrdesc, -) = AArch32\_S1TranslateSD(fault, regime, address[31:0], aligned,
                accdesc);
        end;
    else
        let size : integer = 1;
        (fault, addrdesc) = AArch64\_S1Translate(fault, regime, address, size, aligned, accdesc);
    end;

    if stage == TranslationStage\_12 && fault.statuscode == Fault\_None then
        let slaarch64 : boolean = TRUE;
        if ELUsingAArch32(EL1) && regime == Regime\_EL10 && EL2Enabled() then
            addrdesc.vaddress = ZeroExtend{64}(address);
            (fault, addrdesc) = AArch32\_S2Translate(fault, addrdesc, aligned, accdesc);
        elsif regime == Regime\_EL10 && EL2Enabled() then
            (fault, addrdesc) = AArch64\_S2Translate(fault, addrdesc, slaarch64, aligned, accdesc);
        end;
    end;

    let is_ATS1Ex : boolean = stage != TranslationStage\_12;
    if fault.statuscode != Fault\_None then
        addrdesc = CreateFaultyAddressDescriptor(fault);
        // Take an exception on:
        // * A Synchronous External abort occurs on translation table walk
        // * A stage 2 fault occurs on a stage 1 walk
        // * A GPC Exception (FEAT_RME)
        // * A GPF from ATS1E{1,0}* when executed from EL1 and HCR_EL2.GPF == '1' (FEAT_RME)
        if (IsExternalAbort(fault) ||
            (PSTATE.EL == EL1 && fault.s2fslwalk) ||
            (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF\_None && (
                ReportAsGPCEXception(fault) ||
                (EL2Enabled() && HCR_EL2().GPF == '1' && PSTATE.EL == EL1 && el IN {EL1, EL0} &&
                    is_ATS1Ex)
            ))) then
            if IsFeatureImplemented(FEAT_D128) then
                PAR_EL1() = ARBITRARY : bits(128);
            else
                PAR_EL1()[63:0] = ARBITRARY : bits(64);
            end;
            AArch64\_Abort(addrdesc.fault);
        end;
    end;

    AArch64\_EncodePAR(regime, el, is_ATS1Ex, addrdesc);
    return;
end;
```

Library pseudocode for aarch64/functions/at/AArch64_EncodePAR

```
// AArch64_EncodePAR()
// =====
// Encode PAR register with result of translation.

func AArch64_EncodePAR(regime : Regime, el : bits(2), is_ATS1Ex : boolean,
    addrdesc : AddressDescriptor)
begin
    let paspace : PASpace = addrdesc.paddress.paspace;
    if IsFeatureImplemented(FEAT_D128) then
        PAR_EL1() = Zeros{128};
        if AArch64\_isPARFormatD128(regime, is_ATS1Ex) then
            PAR_EL1().D128 = '1';
        else
            PAR_EL1().D128 = '0';
        end;
    else
        PAR_EL1()[63:0] = Zeros{64};
    end;

    if !IsFault(addrdesc) then
        PAR_EL1().F = '0';
        if IsFeatureImplemented(FEAT_RME) then
            if regime == Regime\_EL3 then
                case paspace of
                    when PAS\_Secure => PAR_EL1().[NSE,NS] = '00';
                    when PAS\_NonSecure => PAR_EL1().[NSE,NS] = '01';
                    when PAS\_Root => PAR_EL1().[NSE,NS] = '10';
                    when PAS\_Realm => PAR_EL1().[NSE,NS] = '11';
                end;

                elsif SecurityStateForRegime(regime) == SS\_Secure then
                    PAR_EL1().NSE = ARBITRARY : bit;
                    PAR_EL1().NS = if paspace == PAS\_Secure then '0' else '1';

                elsif SecurityStateForRegime(regime) == SS\_Realm then
                    if regime == Regime\_EL10 && is_ATS1Ex then
                        PAR_EL1().NSE = ARBITRARY : bit;
                        PAR_EL1().NS = ARBITRARY : bit;
                    else
                        PAR_EL1().NSE = ARBITRARY : bit;
                        PAR_EL1().NS = if paspace == PAS\_Realm then '0' else '1';
                    end;

                else
                    PAR_EL1().NSE = ARBITRARY : bit;
                    PAR_EL1().NS = ARBITRARY : bit;
                end;
            else
                PAR_EL1()[11] = '1'; // RES1
                if SecurityStateForRegime(regime) == SS\_Secure then
                    PAR_EL1().NS = if paspace == PAS\_Secure then '0' else '1';
                else
                    PAR_EL1().NS = ARBITRARY : bit;
                end;
            end;
        end;
        PAR_EL1().SH = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        if IsFeatureImplemented(FEAT_D128) && PAR_EL1().D128 == '1' then
            PAR_EL1()[119:76] = addrdesc.paddress.address[55:12];
        else
            PAR_EL1()[55:12] = addrdesc.paddress.address[55:12];
        end;
        PAR_EL1().ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1()[10] = ImpDefBit("Non-Faulting PAR");
    else
        PAR_EL1().F = '1';
        PAR_EL1().DirtyBit = if addrdesc.fault.dirtybit then '1' else '0';
        PAR_EL1().Overlay = if addrdesc.fault.overlay then '1' else '0';
        PAR_EL1().TopLevel = if addrdesc.fault.toplevel then '1' else '0';
        PAR_EL1().AssuredOnly = if addrdesc.fault.assuredonly then '1' else '0';
        PAR_EL1().FST = AArch64\_PARFaultStatus(addrdesc.fault);
        PAR_EL1().PTW = if addrdesc.fault.s2fslwalk then '1' else '0';
        PAR_EL1().S = if addrdesc.fault.secondstage then '1' else '0';
        PAR_EL1()[11] = '1'; // RES1
        PAR_EL1()[63:48] = ImpDefBits{16}("Faulting PAR");
    end;
    return;
end;
```


Library pseudocode for aarch64/functions/at/AArch64_PARFaultStatus

```
// AArch64_PARFaultStatus()
// =====
// Fault status field decoding of 64-bit PAR.

func AArch64_PARFaultStatus(fault : FaultRecord) => bits(6)
begin
    var fst : bits(6);

    if fault.statuscode == Fault\_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst[1:0] = if fault.level == 1 then '01' else '10';
        fst[5:2] = '1111';
    else
        fst = EncodeLDFSC(fault.statuscode, fault.level);
    end;
    return fst;
end;
```

Library pseudocode for aarch64/functions/at/AArch64_isPARFormatD128

```
// AArch64_isPARFormatD128()
// =====
// Check if last stage of translation uses VMSAv9-128.
// Last stage of translation is stage 2 if enabled, else it is stage 1.

func AArch64_isPARFormatD128(regime : Regime, is_ATS1Ex : boolean) => boolean
begin
    var isPARFormatD128 : boolean;
    // Regime_EL2 does not support VMSAv9-128
    if regime == Regime\_EL2 || !IsFeatureImplemented(FEAT_D128) then
        isPARFormatD128 = FALSE;
    else
        isPARFormatD128 = FALSE;
        case regime of
            when Regime\_EL3 =>
                isPARFormatD128 = TCR_EL3().D128 == '1';
            when Regime\_EL20 =>
                isPARFormatD128 = IsTCR2EL2Enabled() && TCR2_EL2().D128 == '1';
            when Regime\_EL10 =>
                if is_ATS1Ex || !EL2Enabled() || HCR_EL2().[VM,DC] == '00' then
                    isPARFormatD128 = IsTCR2EL1Enabled() && TCR2_EL1().D128 == '1';
                else
                    isPARFormatD128 = VTCR_EL2().D128 == '1';
                end;
        end;
    end;
    return isPARFormatD128;
end;
```

Library pseudocode for aarch64/functions/at/GetPAR_EL1_D128

```
// GetPAR_EL1_D128()
// =====
// Query the PAR_EL1.D128 field

func GetPAR_EL1_D128() => bit
begin
    return if IsFeatureImplemented(FEAT_D128) then PAR_EL1().D128 else '0';
end;
```

Library pseudocode for aarch64/functions/at/GetPAR_EL1_F

```
// GetPAR_EL1_F()
// =====
// Query the PAR_EL1.F field.

func GetPAR_EL1_F() => bit
begin
    var F : bit;

    F = PAR_EL1().F;
    return F;
end;
```

Library pseudocode for aarch64/functions/barrierop/MemBarrierOp

```
// MemBarrierOp
// =====
// Memory barrier instruction types.

type MemBarrierOp of enumeration {
    MemBarrierOp_DSB,      // Data Synchronization Barrier
    MemBarrierOp_DMB,      // Data Memory Barrier
    MemBarrierOp_ISB,      // Instruction Synchronization Barrier
    MemBarrierOp_SSBB,      // Speculative Synchronization Barrier to VA
    MemBarrierOp_PSSBB,     // Speculative Synchronization Barrier to PA
    MemBarrierOp_SB         // Speculation Barrier
};
```

Library pseudocode for aarch64/functions/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// =====
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

func BFXPreferred(sf : bit, uns : bit, imms : bits(6), immr : bits(6)) => boolean
begin
    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;
    end;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf::'11111' then
        return FALSE;
    end;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        end;
        // must not match 64-bit SXT[BHW]
        if sf::uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;
        end;
    end;

    // must be UBFX/SBFX alias
    return TRUE;
end;
```



```

// AltDecodeBitMasks()
// =====
// Alternative but logically equivalent implementation of DecodeBitMasks() that
// uses simpler primitives to compute tmask and wmask.

func AltDecodeBitMasks{M}(immN : bit, imms : bits(6), immr : bits(6),
                          immediate : boolean) => (bits(M), bits(M))
begin
  var tmask, wmask : bits(64);
  var tmask_and, wmask_and : bits(6);
  var tmask_or, wmask_or : bits(6);
  var levels : bits(6);

  // Compute log2 of element size
  // 2^len must be in range [2, M]
  if immN::NOT(imms) == '000000x' then Undefined(); end;
  let len : integer{} = HighestSetBitNZ(immN::NOT(imms));
  assert 2 <= (2^len) && (2^len) <= M;

  // Determine s, r and s - r parameters
  levels = ZeroExtend{6}(Ones{len});

  // For logical immediates an all-ones value of s is reserved
  // since it would generate a useless all-ones result (many times)
  if immediate && (imms AND levels) == levels then Undefined(); end;

  let s : integer{} = UInt(imms AND levels);
  let r : integer{} = UInt(immr AND levels);
  let diff : integer{} = s - r;    // 6-bit subtract with borrow

  // Compute "top mask"
  tmask_and = diff[5:0] OR NOT(levels);
  tmask_or  = diff[5:0] AND levels;

  tmask = Ones{64};
  tmask = ((tmask
    AND Replicate{64, 2}(Replicate{1}(tmask_and[0]) :: Ones{1}))
    OR Replicate{64, 2}(Zeros{1} :: Replicate{1}(tmask_or[0])));
  // optimization of first step:
  // tmask = Replicate{64}(tmask_and[0] :: '1');
  tmask = ((tmask
    AND Replicate{64, 4}(Replicate{2}(tmask_and[1]) :: Ones{2}))
    OR Replicate{64, 4}(Zeros{2} :: Replicate{2}(tmask_or[1])));
  tmask = ((tmask
    AND Replicate{64, 8}(Replicate{4}(tmask_and[2]) :: Ones{4}))
    OR Replicate{64, 8}(Zeros{4} :: Replicate{4}(tmask_or[2])));
  tmask = ((tmask
    AND Replicate{64, 16}(Replicate{8}(tmask_and[3]) :: Ones{8}))
    OR Replicate{64, 16}(Zeros{8} :: Replicate{8}(tmask_or[3])));
  tmask = ((tmask
    AND Replicate{64, 32}(Replicate{16}(tmask_and[4]) :: Ones{16}))
    OR Replicate{64, 32}(Zeros{16} :: Replicate{16}(tmask_or[4])));
  tmask = ((tmask
    AND Replicate{64, 64}(Replicate{32}(tmask_and[5]) :: Ones{32}))
    OR Replicate{64, 64}(Zeros{32} :: Replicate{32}(tmask_or[5])));

  // Compute "wraparound mask"
  wmask_and = immr OR NOT(levels);
  wmask_or  = immr AND levels;

  wmask = Zeros{64};
  wmask = ((wmask
    AND Replicate{64, 2}(Ones{1} :: Replicate{1}(wmask_and[0]))
    OR Replicate{64, 2}(Replicate{1}(wmask_or[0]) :: Zeros{1}));
  // wmask = Replicate{64}(wmask_or[0] :: '0');
  wmask = ((wmask
    AND Replicate{64, 4}(Ones{2} :: Replicate{2}(wmask_and[1]))
    OR Replicate{64, 4}(Replicate{2}(wmask_or[1]) :: Zeros{2}));
  wmask = ((wmask
    AND Replicate{64, 8}(Ones{4} :: Replicate{4}(wmask_and[2]))
    OR Replicate{64, 8}(Replicate{4}(wmask_or[2]) :: Zeros{4}));
  wmask = ((wmask
    AND Replicate{64, 16}(Ones{8} :: Replicate{8}(wmask_and[3]))
    OR Replicate{64, 16}(Replicate{8}(wmask_or[3]) :: Zeros{8}));
  wmask = ((wmask
    AND Replicate{64, 32}(Ones{16} :: Replicate{16}(wmask_and[4]))
    OR Replicate{64, 32}(Replicate{16}(wmask_or[4]) :: Zeros{16}));
  wmask = ((wmask
    AND Replicate{64, 64}(Ones{32} :: Replicate{32}(wmask_and[5]))
    OR Replicate{64, 64}(Replicate{32}(wmask_or[5]) :: Zeros{32}));

  if diff[6] != '0' then // borrow from s - r
    wmask = wmask AND tmask;
  else
    wmask = wmask OR tmask;
  end;

  return (wmask[M-1:0], tmask[M-1:0]);
end;

```

Library pseudocode for aarch64/functions/bitmasks/DecodeBitMasks

```
// DecodeBitMasks()
// =====
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

func DecodeBitMasks{M}(immN : bit, imms : bits(6), immr : bits(6),
    immediate : boolean) => (bits(M), bits(M))
begin
    var levels : bits(6);

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    if immN::NOT(imms) == '000000x' then Undefined(); end;
    let len : integer{} = HighestSetBitNZ(immN::NOT(imms));
    assert 2 <= (2^len) && (2^len) <= M;

    // Determine s, r and s - r parameters
    levels = ZeroExtend{6}(Ones{len});

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then Undefined(); end;

    let s : integer{} = UInt(imms AND levels);
    let r : integer{} = UInt(immr AND levels);
    let diff : integer{} = s - r;    // 6-bit subtract with borrow

    let esize : integer{} = 1 << len;
    let d : integer{} = UInt(diff[len-1:0]);
    let welem : bits(esize) = ZeroExtend{}(Ones{s + 1});
    let telem : bits(esize) = ZeroExtend{}(Ones{d + 1});
    let wmask : bits(M) = Replicate{}(ROR(welem, r));
    let tmask : bits(M) = Replicate{}(telem);
    return (wmask, tmask);
end;
```

Library pseudocode for aarch64/functions/cache/AArch64_DataMemZero

```
// AArch64_DataMemZero()
// =====
// Write Zero to data memory.

func AArch64_DataMemZero(regval : bits(64), vaddress : bits(64),
                        accdesc_in : AccessDescriptor, size : integer)
begin
    var accdesc : AccessDescriptor = accdesc_in;

    // If the instruction targets tags as a payload, confer with system register configuration
    // which may override this.
    if accdesc.tagaccess then
        accdesc.tagaccess = IsMTEEnabled(accdesc.el);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64\_AccessIsTagChecked(vaddress, accdesc);
    end;

    let aligned : boolean = TRUE;
    var memaddrdesc : AddressDescriptor = AArch64\_TranslateAddress(vaddress,
                                                                accdesc, aligned, size);

    if IsFault(memaddrdesc) then
        if !IsDebugException(memaddrdesc.fault) then
            memaddrdesc.fault.vaddress = regval;
        end;
        AArch64\_Abort(memaddrdesc.fault);
    end;

    for i = 0 to size-1 do
        if accdesc.tagchecked then
            let ltag : bits(4) = AArch64\_LogicalAddressTag(vaddress);
            let bytes : integer = 1;
            let readcheck : boolean = FALSE;
            var fault : FaultRecord = AArch64\_CheckTag(memaddrdesc, accdesc, readcheck,
                                                    bytes, ltag);

            if fault.statuscode != Fault\_None then
                if ImpDefBool(
                    "DC_ZVA tag fault reported with lowest faulting address") then
                    AArch64\_Abort(fault);
                else
                    fault.vaddress = regval;
                    AArch64\_Abort(fault);
                end;
            end;
        end;
        let memstatus : PhysMemRetStatus = PhysMemWrite{8}(memaddrdesc, accdesc, Zeros{8});
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        end;

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    end;
    return;
end;
```

Library pseudocode for aarch64/functions/cache/AArch64_WriteTagMem

```
// AArch64_WriteTagMem()
// =====
// Write to tag memory.

func AArch64_WriteTagMem(regval : bits(64), cachetype : CacheType,
                        accdesc_in : AccessDescriptor, size : integer)
begin
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    var accdesc : AccessDescriptor = accdesc_in;
    let vaddress : bits(64) = AlignDownSize{64}(regval, size as integer{16..2048});
    let count : integer = size >> LOG2\_TAG\_GRANULE;
    let tag : bits(4) = AArch64\_AllocationTagFromAddress(vaddress);
    let aligned : boolean = IsAlignedSize{64}(vaddress, TAG\_GRANULE);
    assert aligned;

    accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    var (memtagtype, memaddrdesc) = AArch64\_TranslateTagAddress(vaddress, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        if IsDebugException(memaddrdesc.fault) then
            AArch64\_Abort(memaddrdesc.fault);
        else
            memaddrdesc.fault.vaddress = regval;
            AArch64\_Abort(memaddrdesc.fault);
        end;
    end;

    if !accdesc.tagaccess || memtagtype != MemTag\_AllocationTagged then
        return;
    end;

    var memstatus : PhysMemRetStatus;

    for i = 0 to count-1 do
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, tag);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        end;

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + TAG\_GRANULE;
        memaddrdesc.vaddress = memaddrdesc.vaddress + TAG\_GRANULE;
    end;

    return;
end;
```

Library pseudocode for aarch64/functions/compareop/CompareOp

```
// CompareOp
// =====
// Vector compare instruction types.

type CompareOp of enumeration {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                              CompareOp_LE, CompareOp_LT};
```

Library pseudocode for aarch64/functions/countop/CountOp

```
// CountOp
// =====
// Bit counting instruction types.

type CountOp of enumeration {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

Library pseudocode for aarch64/functions/cpa/EffectiveCPTA

```
// EffectiveCPTA()
// =====
// Returns the CPTA bit applied to Checked Pointer Arithmetic for Addition in the given EL.

func EffectiveCPTA(el : bits(2)) => bit
begin
    if !IsFeatureImplemented(FEAT_CPA2) then
        return '0';
    end;

    if Halted\(\) then
        return '0';
    end;

    var cpta : bits(1);
    let regime : Regime = TranslationRegime(el);

    case regime of
        when Regime\_EL3 =>
            cpta = SCTLR2_EL3().CPTA;
        when Regime\_EL2 =>
            if IsSCTLR2EL2Enabled\(\) then
                cpta = SCTLR2_EL2().CPTA;
            else
                cpta = '0';
            end;
        when Regime\_EL20 =>
            if IsSCTLR2EL2Enabled\(\) then
                cpta = if el == EL0 then SCTLR2_EL2().CPTA0 else SCTLR2_EL2().CPTA;
            else
                cpta = '0';
            end;
        when Regime\_EL10 =>
            if IsSCTLR2EL1Enabled\(\) then
                cpta = if el == EL0 then SCTLR2_EL1().CPTA0 else SCTLR2_EL1().CPTA;
            else
                cpta = '0';
            end;
        otherwise =>
            unreachable;
    end;

    return cpta;
end;
```


Library pseudocode for aarch64/functions/cpa/EffectiveCPTM

```
// EffectiveCPTM()
// =====
// Returns the CPTM bit applied to Checked Pointer Arithmetic for Multiplication in the given EL.

func EffectiveCPTM(el : bits(2)) => bit
begin
    if !IsFeatureImplemented(FEAT_CPA2) then
        return '0';
    end;

    if EffectiveCPTA(el) == '0' then
        return '0';
    end;

    if Halted() then
        return '0';
    end;

    var cptm : bits(1);
    let regime : Regime = TranslationRegime(el);

    case regime of
        when Regime\_EL3 =>
            cptm = SCTLR2_EL3().CPTM;
        when Regime\_EL2 =>
            if IsSCTLR2EL2Enabled() then
                cptm = SCTLR2_EL2().CPTM;
            else
                cptm = '0';
            end;
        when Regime\_EL20 =>
            if IsSCTLR2EL2Enabled() then
                cptm = if el == EL0 then SCTLR2_EL2().CPTM0 else SCTLR2_EL2().CPTM;
            else
                cptm = '0';
            end;
        when Regime\_EL10 =>
            if IsSCTLR2EL1Enabled() then
                cptm = if el == EL0 then SCTLR2_EL1().CPTM0 else SCTLR2_EL1().CPTM;
            else
                cptm = '0';
            end;
        otherwise =>
            unreachable;
    end;

    return cptm;
end;
```

Library pseudocode for aarch64/functions/cpa/PointerAddCheck

```
// PointerAddCheck()
// =====
// Apply Checked Pointer Arithmetic for addition.

func PointerAddCheck(result : bits(64), base : bits(64)) => bits(64)
begin
    return PointerCheckAtEL(PSTATE.EL, result, base, FALSE);
end;
```

Library pseudocode for aarch64/functions/cpa/PointerAddCheckAtEL

```
// PointerAddCheckAtEL()
// =====
// Apply Checked Pointer Arithmetic for addition at the specified EL.

func PointerAddCheckAtEL(el : bits(2), result : bits(64), base : bits(64)) => bits(64)
begin
    return PointerCheckAtEL(el, result, base, FALSE);
end;
```

Library pseudocode for aarch64/functions/cpa/PointerCheckAtEL

```
// PointerCheckAtEL()
// =====
// Apply Checked Pointer Arithmetic at the specified EL.

func PointerCheckAtEL(el : bits(2), result : bits(64), base : bits(64),
    cptm_detected : boolean) => bits(64)
begin
    var rv : bits(64) = result;

    let previous_detection : boolean = (base[55] != base[54]);
    let cpta_detected : boolean = (result[63:56] != base[63:56] || previous_detection);

    if ((cpta_detected && EffectiveCPTA(el) == '1') ||
        (cptm_detected && EffectiveCPTM(el) == '1')) then
        rv[63:55] = base[63:55];
        rv[54] = NOT(rv[55]);
    end;

    return rv;
end;
```

Library pseudocode for aarch64/functions/cpa/PointerMultiplyAddCheck

```
// PointerMultiplyAddCheck()
// =====
// Apply Checked Pointer Arithmetic for multiplication.

func PointerMultiplyAddCheck(result : bits(64), base : bits(64),
    cptm_detected : boolean) => bits(64)
begin
    return PointerCheckAtEL(PSTATE.EL, result, base, cptm_detected);
end;
```

Library pseudocode for aarch64/functions/d128/IsD128Enabled

```
// IsD128Enabled()
// =====
// Returns true if 128-bit page descriptor is enabled

func IsD128Enabled(el : bits(2)) => boolean
begin
    var d128enabled : boolean;
    if IsFeatureImplemented(FEAT_D128) then
        case el of
            when EL0 =>
                if !ELIsInHost(EL0) then
                    d128enabled = IsTCR2EL1Enabled() && TCR2_EL1().D128 == '1';
                else
                    d128enabled = IsTCR2EL2Enabled() && TCR2_EL2().D128 == '1';
                end;
            when EL1 =>
                d128enabled = IsTCR2EL1Enabled() && TCR2_EL1().D128 == '1';
            when EL2 =>
                d128enabled = IsTCR2EL2Enabled() && IsInHost() && TCR2_EL2().D128 == '1';
            when EL3 =>
                d128enabled = TCR_EL3().D128 == '1';
        end;
    else
        d128enabled = FALSE;
    end;

    return d128enabled;
end;
```

Library pseudocode for aarch64/functions/dc/AArch64_CanTrapDC

```
// AArch64_CanTrapDC()
// =====
// Determines whether the execution of the DC instruction can be trapped.

func AArch64_CanTrapDC(cachetype : CacheType, cacheop : CacheOp, opscope : CacheOpScope) => boolean
begin
    return (!AArch64\_TreatDCAsNOP(cachetype, cacheop, opscope) || ImpDefBool(
        "When DC is treated as NOP, data cache maintenance operations are trapped"));
end;
```



```

// AArch64_DC()
// =====
// Perform Data Cache Operation.

func AArch64_DC(regval : bits(64), cachetype : CacheType,
                cacheop : CacheOp, opscope_in : CacheOpScope)
begin
    var opscope : CacheOpScope = opscope_in;
    var cache : CacheRecord;

    cache.acctype = AccessType\_DC;
    cache.cachetype = cachetype;
    cache.cacheop = cacheop;
    cache.opscope = opscope;

    if opscope == CacheOpScope\_SetWay then
        let ss : SecurityState = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        cache.shareability = Shareability\_NSH;
        cache.(setnum, waynum, level) = DecodeSW(regval, cachetype);
        if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            (HCR\_EL2().SWIO == '1' || HCR\_EL2().[DC,VM] != '00')) then
            cache.cacheop = CacheOp\_CleanInvalidate;
        end;

        CACHE\_OP(cache);
        return;
    end;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid = VMID();
        else
            cache.is_vmid_valid = FALSE;
        end;
    else
        cache.is_vmid_valid = FALSE;
    end;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid = ASID();
    else
        cache.is_asid_valid = FALSE;
    end;

    if (opscope == CacheOpScope\_PoPS &&
        ImpDefBool("Memory system does not support PoPS")) then
        opscope = CacheOpScope\_PoC;
    end;
    if (opscope == CacheOpScope\_PoDP &&
        ImpDefBool("Memory system does not support PoDP")) then
        opscope = CacheOpScope\_PoP;
    end;
    if (opscope == CacheOpScope\_PoP &&
        ImpDefBool("Memory system does not support PoP")) then
        opscope = CacheOpScope\_PoC;
    end;
    var vaddress : bits(64) = regval;

    var size : integer = 0; // by default no watchpoint address
    if cacheop == CacheOp\_Invalidate then
        size = DataCacheWatchpointSize();
        vaddress = AlignDownSize{64}(regval, size as AddressSize);
    end;

    if opscope == CacheOpScope\_PoE then
        cache.shareability = Shareability\_OSH;
        cache.paddress.address = regval[NUM\_PABITS-1:0];
        let nse2 : bit = if IsFeatureImplemented(FEAT\_RME\_GDI) then regval[61] else '0';
        cache.paddress.paspace = DecodePASpace(nse2, regval[62], regval[63]);
        cache.cpas = CPASAtPAS(cache.paddress.paspace);

        // If a Reserved encoding is selected, the instruction is permitted to be treated as a NOP.
        if cache.paddress.paspace != PAS\_Realm then
            ExecuteAsNOP();
        end;

        if ImpDefBool("Apply granule protection check on DC to PoE") then
            var memaddrdesc : AddressDescriptor;
            let accdesc : AccessDescriptor = CreateAccDescDC(cache);
            memaddrdesc.paddress = cache.paddress;
            memaddrdesc.fault.gpcf = GranuleProtectionCheck(memaddrdesc, accdesc);

            if memaddrdesc.fault.gpcf.gpf != GPCF\_None then
                memaddrdesc.fault.statuscode = Fault\_GPCFOnOutput;
                memaddrdesc.fault.paddress = memaddrdesc.paddress;
                memaddrdesc.fault.vaddress = ARBITRARY : bits(64);
            end;
        end;
    end;
end;

```

```

        AArch64\_Abort(memaddrdesc.fault);
    end;
end;
elseif opscope == CacheOpScope\_PoPA then
    cache.shareability = Shareability\_OSH;
    cache.paddress.address = regval[NUM\_PABITS-1:0];
    let nse2 : bit = if IsFeatureImplemented(FEAT_RME_GDI) then regval[61] else '0';
    cache.paddress.paspace = DecodePASpace(nse2, regval[62], regval[63]);

    // If {NSE2, NSE, NS} is reserved, then no cache entries are required
    // to be cleaned or invalidated.
    if cache.paddress.paspace IN {PAS\_NA6, PAS\_NA7} then
        ExecuteAsNOP();
    end;
    cache.cpas = CPASAtPAS(cache.paddress.paspace);
else
    cache.vaddress = vaddress;
    let aligned : boolean = TRUE;
    let accdesc : AccessDescriptor = CreateAccDescDC(cache);
    var memaddrdesc : AddressDescriptor = AArch64\_TranslateAddress(vaddress, accdesc,
                                                                aligned, size);

    if IsFault(memaddrdesc) then
        memaddrdesc.fault.vaddress = regval;
        AArch64\_Abort(memaddrdesc.fault);
    end;

    cache.paddress = memaddrdesc.paddress;
    cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);

    if (opscope IN {
        CacheOpScope\_PoDP,
        CacheOpScope\_PoP,
        CacheOpScope\_PoC,
        CacheOpScope\_PoU
    }) then
        cache.shareability = memaddrdesc.memattrs.shareability;
    else
        cache.shareability = Shareability\_NSH;
    end;
end;

if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
    HCR\_EL2().[DC,VM] != '00') then
    cache.cacheop = CacheOp\_CleanInvalidate;
end;

// If Secure state is not implemented, but RME is, the instruction acts as a NOP
if cache.cpas == CPAS\_Secure && !HaveSecureState() then
    return;
end;

CACHE\_OP(cache);
return;
end;

```

Library pseudocode for aarch64/functions/dc/AArch64_MemZero

```

// AArch64_MemZero()
// =====

func AArch64_MemZero(regval : bits(64), cachetype : CacheType)
begin
    let size : integer{} = 4*(2^(UInt(DCZID_EL0().BS)));
    assert size <= MAX\_ZERO\_BLOCK\_SIZE;
    if IsFeatureImplemented(FEAT_MTE2) then
        assert size >= TAG\_GRANULE;
    end;
    let vaddress : bits(64) = AlignDownSize{}(regval, size);

    let accdesc : AccessDescriptor = CreateAccDescDCZero(cachetype);

    if cachetype != CacheType\_Data then
        AArch64\_WriteTagMem(regval, cachetype, accdesc, size);
    end;

    if cachetype IN {CacheType\_Data, CacheType\_Data\_Tag} then
        AArch64\_DataMemZero(regval, vaddress, accdesc, size);
    end;
    return;
end;

```

Library pseudocode for aarch64/functions/dc/AArch64_TreatDCAsNOP

```
// AArch64_TreatDCAsNOP()
// =====
// Determines whether the execution of the DC instruction is treated as a NOP.

func AArch64_TreatDCAsNOP(cachetype : CacheType, cacheop : CacheOp,
                           opscope : CacheOpScope) => boolean
begin
    // DC to PoU: IMPLEMENTATION DEFINED - treated as NOP if LoUU and LoUIS are 0
    if opscope == CacheOpScope\_PoU && CLIDR_EL1().LoUU == '000' && CLIDR_EL1().LoUIS == '000' then
        return ImpDefBool("DC to PoU is treated as a NOP");
    end;

    // DC to PoC: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0
    if opscope == CacheOpScope\_PoC && CLIDR_EL1().LoC == '000' then
        return ImpDefBool("DC to PoC is treated as a NOP");
    end;

    // DC to PoP: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0 and PoP unsupported
    if (opscope == CacheOpScope\_PoP && CLIDR_EL1().LoC == '000' &&
        !ImpDefBool("Memory system supports PoP")) then
        return ImpDefBool("DC to PoP is treated as a NOP");
    end;

    // DC to PoDP: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0 and PoP/PoDP unsupported
    if (opscope == CacheOpScope\_PoDP && CLIDR_EL1().LoC == '000' &&
        !ImpDefBool("Memory system supports PoP") &&
        !ImpDefBool("Memory system supports PoDP")) then
        return ImpDefBool("DC to PoDP is treated as a NOP");
    end;

    return FALSE;
end;
```

Library pseudocode for aarch64/functions/dc/MemZero

```
// MemZero block size
// =====

constant MAX_ZERO_BLOCK_SIZE : integer = 2048;
```

Library pseudocode for aarch64/functions/eret/AArch64_ExceptionReturn

```
// AArch64_ExceptionReturn()
// =====

func AArch64_ExceptionReturn(new_pc_in : bits(64), spsr : bits(64))
begin
    var new_pc : bits(64) = new_pc_in;

    if IsFeatureImplemented(FEAT_IESB) then
        var sync_errors : boolean = SCTLR\_ELx\(\).IESB == '1';
        if IsFeatureImplemented(FEAT_DoubleFault) then
            sync_errors = sync_errors || (SCR\_EL3\(\).\[EA,NMEA\] == '11' && PSTATE.EL == EL3);
        end;
        if sync_errors then
            SynchronizeErrors();
            let iesb_req : boolean = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
        end;
    end;

    var brbe_source_allowed : boolean = FALSE;
    var brbe_source_address : bits(64) = Zeros{};
    if IsFeatureImplemented(FEAT_BRBE) then
        brbe_source_allowed = BranchRecordAllowed\(PSTATE.EL\);
        brbe_source_address = PC64\(\);
    end;

    if !IsFeatureImplemented(FEAT_ExS) || SCTLR\_ELx\(\).EOS == '1' then
        SynchronizeContext();
    end;

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    let source_el : bits(2) = PSTATE.EL;
    let illegal_psr_state : boolean = IllegalExceptionReturn{64}(spsr);
    SetPSTATEFromPSR{64}(spsr, illegal_psr_state);
    ClearExclusiveLocal\(ProcessorID\(\)\);
    SendEventLocal\(\);

    if illegal_psr_state && spsr[4] == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc[63:32] = ARBITRARY : bits(32);
        new_pc[1:0] = ARBITRARY : bits(2);
    elseif UsingAArch32\(\) then // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc[0] = '0'; // T32
        else
            new_pc[1:0] = '00'; // A32
        end;
    else // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64 BranchAddr(new_pc, PSTATE.EL);
    end;

    if IsFeatureImplemented(FEAT_BRBE) then
        BRBEExceptionReturn(new_pc, source_el,
            brbe_source_allowed, brbe_source_address);
    end;

    if UsingAArch32\(\) then
        if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then ResetsVEState\(\); end;

        // 32 most significant bits are ignored.
        let branch_conditional : boolean = FALSE;
        BranchTo{32}(new_pc[31:0], BranchType ERET, branch_conditional);
    else
        BranchToAddr{64}(new_pc, BranchType ERET);
    end;

    CheckExceptionCatch(FALSE); // Check for debug event on exception return
end;
```

Library pseudocode for aarch64/functions/exclusive/AArch64_ExclusiveMonitorsPass

```
// AArch64_ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give a memory abort.

func AArch64_ExclusiveMonitorsPass(address : bits(64), size : integer{1, 2, 4, 8, 16, 32},
                                   accdesc : AccessDescriptor) => boolean
begin
    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned && AArch64\_UnalignedAccessFaults(accdesc, address, size) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64\_Abort(fault);
    end;

    if !AArch64\_IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;
    end;

    let memaddrdesc : AddressDescriptor = AArch64\_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64\_Abort(memaddrdesc.fault);
    end;

    var passed : boolean = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability\_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    return passed;
end;
```

Library pseudocode for aarch64/functions/exclusive/AArch64_IsExclusiveVA

```
// AArch64_IsExclusiveVA()
// =====
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

impdef func AArch64_IsExclusiveVA(address : bits(64),
                                   processorid : integer,
                                   size : integer) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for aarch64/functions/exclusive/AArch64_MarkExclusiveVA

```
// AArch64_MarkExclusiveVA()
// =====
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

func AArch64_MarkExclusiveVA(address : bits(64), processorid : integer, size : integer)
begin
    return;
end;
```


Library pseudocode for aarch64/functions/exclusive/AArch64_SetExclusiveMonitors

```
// AArch64_SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

func AArch64_SetExclusiveMonitors(address : bits(64), size : integer{1, 2, 4, 8, 16, 32},
                                accdesc : AccessDescriptor)
begin
    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned && AArch64\_UnalignedAccessFaults(accdesc, address, size) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64\_Abort(fault);
    end;

    let memaddrdesc : AddressDescriptor = AArch64\_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;
    end;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64\_MarkExclusiveVA(address, ProcessorID(), size);
end;
```

Library pseudocode for aarch64/functions/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =====
// Decode a register extension option

func DecodeRegExtend(op : bits(3)) => ExtendType
begin
    case op of
        when '000' => return ExtendType\_UXTB;
        when '001' => return ExtendType\_UXTH;
        when '010' => return ExtendType\_UXTW;
        when '011' => return ExtendType\_UXTX;
        when '100' => return ExtendType\_SXTB;
        when '101' => return ExtendType\_SXTH;
        when '110' => return ExtendType\_SXTW;
        when '111' => return ExtendType\_SXTX;
    end;
end;
```

Library pseudocode for aarch64/functions/extendreg/ExtendReg

```
// ExtendReg()
// =====
// Perform a register extension and shift

func ExtendReg{N}(reg : integer, exttype : ExtendType, shift : integer{0..4}) => bits(N)
begin
    let val : bits(N) = X{}(reg);
    var unsigned : boolean;
    var len : ESize;

    case exttype of
        when ExtendType\_SXTB => unsigned = FALSE; len = 8;
        when ExtendType\_SXTH => unsigned = FALSE; len = 16;
        when ExtendType\_SXTW => unsigned = FALSE; len = 32;
        when ExtendType\_SXTX => unsigned = FALSE; len = 64;
        when ExtendType\_UXTB => unsigned = TRUE; len = 8;
        when ExtendType\_UXTH => unsigned = TRUE; len = 16;
        when ExtendType\_UXTW => unsigned = TRUE; len = 32;
        when ExtendType\_UXTX => unsigned = TRUE; len = 64;
    end;

    // Sign or zero extend bottom LEN bits of register and shift left by SHIFT
    let nbits : integer{} = Min(len, N) as integer{8..N};
    let extval : bits(N) = Extend{}(val[nbits-1:0], unsigned);
    return LSL(extval, shift);
end;
```

Library pseudocode for aarch64/functions/extendreg/ExtendType

```
// ExtendType
// =====
// AArch64 register extend and shift.

type ExtendType of enumeration {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SCTX,
                                ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UCTX};
```

Library pseudocode for aarch64/functions/fpconvop/FPConvOp

```
// FPConvOp
// =====
// Floating-point convert/move instruction types.

type FPConvOp of enumeration {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                              FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF,
                              FPConvOp_CVT_FtoI_JS
};
```

Library pseudocode for aarch64/functions/fpmaxminop/FPMMaxMinOp

```
// FPMMaxMinOp
// =====
// Floating-point min/max instruction types.

type FPMMaxMinOp of enumeration {FPMMaxMinOp_MAX, FPMMaxMinOp_MIN,
                                 FPMMaxMinOp_MAXNUM, FPMMaxMinOp_MINNUM};
```

Library pseudocode for aarch64/functions/fpmr/CheckFPMREnabled

```
// CheckFPMREnabled()
// =====
// Check for Undefined instruction exception on indirect FPMR accesses.

func CheckFPMREnabled()
begin
    assert IsFeatureImplemented(FEAT_FPMR);

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            if SCTL_EL1().EnFPM == '0' then Undefined(); end;
        else
            if SCTL_EL2().EnFPM == '0' then Undefined(); end;
        end;
    end;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        if !IsHCRXEL2Enabled() || HCRX_EL2().EnFPM == '0' then Undefined(); end;
    end;

    if PSTATE.EL != EL3 && HaveEL(EL3) then
        if SCR_EL3().EnFPM == '0' then Undefined(); end;
    end;
end;
```

Library pseudocode for aarch64/functions/fpscale/FP ClampScale

```
// FP ClampScale()
// =====

func FP ClampScale{N}(op : bits(N), scale_in : integer) => integer
begin
  assert N IN {16,32,64};
  var E : integer;
  var exp : integer;
  case N of
    when 16 =>
      E = 5;
      exp = UInt(op[14:10]);
    when 32 =>
      E = 8;
      exp = UInt(op[30:23]);
    when 64 =>
      E = 11;
      exp = UInt(op[62:52]);
    otherwise => unreachable;
  end;

  let F : integer = N - (E + 1);
  let emax : integer = (1 << E) - 1;
  let min_scale : integer = -(F + 1);
  let max_scale : integer = emax + (F + 1);

  let scale : integer = Max(min_scale - exp, Min(scale_in, max_scale - exp));
  return scale;
end;
```

Library pseudocode for aarch64/functions/fpscale/FP Scale

```
// FP Scale()
// =====

func FP Scale{N}(op : bits(N), scale : integer, fpcr : FPCR_Type) => bits(N)
begin
  assert N IN {16,32,64};
  var result : bits(N);
  let (fptype, sign, value) : (FType, bit, real) = FPUnpack{N}(op, fpcr);

  if fptype == FType_SNaN || fptype == FType_QNaN then
    result = FPProcessNaN{N}(fptype, op, fpcr);
  elsif fptype == FType_Zero then
    result = FPZero{N}(sign);
  elsif fptype == FType_Infinity then
    result = FPInfinity{N}(sign);
  else
    let clamped_scale : integer = FP ClampScale{N}(op, scale);
    result = FPRound{N}(value * (2.0^clamped_scale), fpcr);
    FPProcessDenorm(fptype, N, fpcr);
  end;
  return result;
end;
```

Library pseudocode for aarch64/functions/fpunaryop/FPUnaryOp

```
// FPUnaryOp
// =====
// Floating-point unary instruction types.

type FPUnaryOp of enumeration {FPUnaryOp_ABS, FPUnaryOp_MOV,
                               FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// =====

func FPRSqrtStepFused{N}(op1_in : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
  assert N IN {16, 32, 64};
  var fpcr : FPCR_Type = fpcr_in;
  var result : bits(N);
  var op1 : bits(N) = op1_in;
  var done : boolean;
  op1 = FPNeg{N}(op1, fpcr);
  let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
  let fpexc : boolean = !altfp; // Generate no floating-point exceptions
  if altfp then fpcr.[FIZ,FZ] = '11'; end; // Flush denormal input and output to zero
  if altfp then fpcr.RMode = '00'; end; // Use RNE rounding mode

  let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
  let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);
  (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
  let rounding : FPRounding = FPRoundingMode(fpcr);

  if !done then
    let inf1 : boolean = (type1 == FType\_Infinity);
    let inf2 : boolean = (type2 == FType\_Infinity);
    let zero1 : boolean = (type1 == FType\_Zero);
    let zero2 : boolean = (type2 == FType\_Zero);

    if (inf1 && zero2) || (zero1 && inf2) then
      result = FPOnePointFive{N}('0');
    elsif inf1 || inf2 then
      result = FPInfinity{N}(sign1 XOR sign2);
    else
      // Fully fused multiply-add and halve
      let result_value : real = (3.0 + (value1 * value2)) / 2.0;
      if result_value == 0.0 then
        // Sign of exact zero result depends on rounding mode
        let sign : bit = if rounding == FPRounding\_NEGINF then '1' else '0';
        result = FPZero{N}(sign);
      else
        result = FPRound{N}(result_value, fpcr, rounding, fpexc);
      end;
    end;
  end;

  return result;
end;
```

Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// =====

func FPRecipStepFused{N}(op1_in : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16, 32, 64};
    var fpcr : FPCR_Type = fpcr_in;
    var op1 : bits(N) = op1_in;
    var result : bits(N);
    var done : boolean;
    op1 = FPNeg{N}(op1, fpcr);

    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    let fpexc : boolean = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.[FIZ,FZ] = '11'; end; // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; end; // Use RNE rounding mode

    let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
    let rounding : FPRounding = FPRoundingMode(fpcr);

    if !done then
        let inf1 : boolean = (type1 == FType\_Infinity);
        let inf2 : boolean = (type2 == FType\_Infinity);
        let zero1 : boolean = (type1 == FType\_Zero);
        let zero2 : boolean = (type2 == FType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo{N}('0');
        elsif inf1 || inf2 then
            result = FPInfinity{N}(sign1 XOR sign2);
        else
            // Fully fused multiply-add
            let result_value : real = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                let sign : bit = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero{N}(sign);
            else
                result = FPRound{N}(result_value, fpcr, rounding, fpexc);
            end;
        end;
    end;

    return result;
end;
```

Library pseudocode for aarch64/functions/gcs/AddGCSExRecord

```
// AddGCSExRecord()
// =====
// Generates and then writes an exception record to the
// current Guarded Control Stack.

func AddGCSExRecord(elr : bits(64), spsr : bits(64), lr : bits(64))
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp\_STORE, privileged);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem{64}(ptr-8, accdesc) = lr;
    Mem{64}(ptr-16, accdesc) = spsr;
    Mem{64}(ptr-24, accdesc) = elr;
    Mem{64}(ptr-32, accdesc) = Zeros{60}::'1001';

    // Decrement the pointer value
    ptr = ptr - 32;

    SetCurrentGCSPointer(ptr);
    return;
end;
```

Library pseudocode for aarch64/functions/gcs/AddGCSRecord

```
// AddGCSRecord()
// =====
// Generates and then writes a record to the current Guarded
// control stack.

func AddGCSRecord(vaddress : bits(64))
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp\_STORE, privileged);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem{64}(ptr-8, accdesc) = vaddress;

    // Decrement the pointer value
    ptr = ptr - 8;

    SetCurrentGCSPointer(ptr);
    return;
end;
```

Library pseudocode for aarch64/functions/gcs/CheckGCSExRecord

```
// CheckGCSExRecord()
// =====
// Validates the provided values against the top entry of the
// current Guarded Control Stack.

func CheckGCSExRecord(elr : bits(64), spsr : bits(64), lr : bits(64), gcsinst_type : GCSInstruction)
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp\_LOAD, privileged);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    let recorded_first_dword : bits(64) = Mem{64}(ptr, accdesc);
    if recorded_first_dword != Zeros(60)::'1001' then
        GCSDDataCheckException(gcsinst_type);
    end;

    // Check the ELR matches the recorded value
    let recorded_elr : bits(64) = Mem{64}(ptr+8, accdesc);
    if recorded_elr != elr then
        GCSDDataCheckException(gcsinst_type);
    end;

    // Check the SPSR matches the recorded value
    let recorded_spsr : bits(64) = Mem{64}(ptr+16, accdesc);
    if recorded_spsr != spsr then
        GCSDDataCheckException(gcsinst_type);
    end;

    // Check the LR matches the recorded value
    let recorded_lr : bits(64) = Mem{64}(ptr+24, accdesc);
    if recorded_lr != lr then
        GCSDDataCheckException(gcsinst_type);
    end;

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPointer(ptr);
    return;
end;
```

Library pseudocode for aarch64/functions/gcs/CheckGCSSTREnabled

```
// CheckGCSSTREnabled()
// =====
// Trap GCSSTR or GCSSTR instruction if trapping is enabled.

func CheckGCSSTREnabled()
begin
  case PSTATE.EL of
    when EL0 =>
      if GCSCRE0_EL1().STREn == '0' then
        if EL2Enabled() && HCR_EL2().TGE == '1' then
          GCSSTRTrapException(EL2);
        else
          GCSSTRTrapException(EL1);
        end;
      end;
    when EL1 =>
      if GCSCR_EL1().STREn == '0' then
        GCSSTRTrapException(EL1);
      elsif (EL2Enabled() && (!HaveEL(EL3) || SCR_EL3().FGTEn == '1') &&
        HFGITR_EL2().nGCSSTR_EL1 == '0') then
        GCSSTRTrapException(EL2);
      end;
    when EL2 =>
      if GCSCR_EL2().STREn == '0' then
        GCSSTRTrapException(EL2);
      end;
    when EL3 =>
      if GCSCR_EL3().STREn == '0' then
        GCSSTRTrapException(EL3);
      end;
  end;
  return;
end;
```

Library pseudocode for aarch64/functions/gcs/EXLOCKException

```
// EXLOCKException()
// =====
// Handle an EXLOCK exception condition.

func EXLOCKException()
begin
  let preferred_exception_return : bits(64) = ThisInstrAddr{}();
  let vect_offset : integer = 0x0;

  var except : ExceptionRecord = ExceptionSyndrome(Exception\_GCSFail);
  except.syndrome.iss[24] = '0';
  except.syndrome.iss[23:20] = '0001';
  except.syndrome.iss[19:0] = Zeros{20};
  AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/functions/gcs/GCSDataCheckException

```
// GCSDataCheckException()
// =====
// Handle a GCS data check fault condition.

func GCSDataCheckException(gcsinst_type : GCSInstruction)
begin
    var target_el : bits(2);
    let preferred_exception_return : bits(64) = ThisInstrAddr();
    let vect_offset : integer = 0x0;
    var rn_unknown : boolean = FALSE;
    var is_ret : boolean = FALSE;
    var is_reta : boolean = FALSE;

    if PSTATE.EL == EL0 then
        target_el = if (EL2Enabled() && HCR\_EL2\(\).TGE == '1') then EL2 else EL1;
    else
        target_el = PSTATE.EL;
    end;
    var except : ExceptionRecord = ExceptionSyndrome(Exception GCSFail);
    case gcsinst_type of
        when GCSInstType PRET =>
            except.syndrome.iss[4:0] = '00000';
            is_ret = TRUE;
        when GCSInstType POPM =>
            except.syndrome.iss[4:0] = '00001';
        when GCSInstType PRETAA =>
            except.syndrome.iss[4:0] = '00010';
            is_reta = TRUE;
        when GCSInstType PRETAB =>
            except.syndrome.iss[4:0] = '00011';
            is_reta = TRUE;
        when GCSInstType SS1 =>
            except.syndrome.iss[4:0] = '00100';
        when GCSInstType SS2 =>
            except.syndrome.iss[4:0] = '00101';
            rn_unknown = TRUE;
        when GCSInstType POPCX =>
            rn_unknown = TRUE;
            except.syndrome.iss[4:0] = '01000';
        when GCSInstType POPX =>
            except.syndrome.iss[4:0] = '01001';
    end;
    if rn_unknown == TRUE then
        except.syndrome.iss[9:5] = ARBITRARY : bits(5);
    elsif is_ret == TRUE then
        except.syndrome.iss[9:5] = ThisInstr() [9:5];
    elsif is_reta == TRUE then
        except.syndrome.iss[9:5] = '11110';
    else
        except.syndrome.iss[9:5] = ThisInstr() [4:0];
    end;
    except.syndrome.iss[24:10] = Zeros{15};
    except.vaddress = ARBITRARY : bits(64);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/functions/gcs/GCSEnabled

```
// GCSEnabled()
// =====
// Returns TRUE if the Guarded Control Stack is enabled at
// the provided Exception level.

func GCSEnabled(el : bits(2)) => boolean
begin
    if UsingAArch32() then
        return FALSE;
    end;

    if HaveEL(EL3) && el != EL3 && SCR\_EL3\(\).GCSEn == '0' then
        return FALSE;
    end;

    if (el IN {EL0, EL1} && EL2Enabled() && !ELIsInHost(EL0) &&
        (!IsHCRXEL2Enabled() || HCRX\_EL2\(\).GCSEn == '0')) then
        return FALSE;
    end;

    return GCSPCRSSelected(el);
end;
```


Library pseudocode for aarch64/functions/gcs/GCSInstruction

```
// GCSInstruction
// =====

type GCSInstruction of enumeration {
    GCSInstType_PRET,    // Procedure return without Pointer authentication
    GCSInstType_POPM,    // GCSPOPM instruction
    GCSInstType_PRETAA,  // Procedure return with Pointer authentication that used key A
    GCSInstType_PRETAB,  // Procedure return with Pointer authentication that used key B
    GCSInstType_SS1,     // GCSSS1 instruction
    GCSInstType_SS2,     // GCSSS2 instruction
    GCSInstType_POPCX,   // GCSPOPCX instruction
    GCSInstType_POPX     // GCSPOPX instruction
};
```

Library pseudocode for aarch64/functions/gcs/GCSPCREnabled

```
// GCSPCREnabled()
// =====
// Returns TRUE if the Guarded Control Stack is PCR enabled
// at the provided Exception level.

func GCSPCREnabled(el : bits(2)) => boolean
begin
    return GCSPCRSelected(el) && GCSEnabled(el);
end;
```

Library pseudocode for aarch64/functions/gcs/GCSPCRSelected

```
// GCSPCRSelected()
// =====
// Returns TRUE if the Guarded Control Stack is PCR selected
// at the provided Exception level.

func GCSPCRSelected(el : bits(2)) => boolean
begin
    case el of
        when EL0 => return GCSCRE0\_EL1().PCRSEL == '1';
        when EL1 => return GCSCR\_EL1().PCRSEL == '1';
        when EL2 => return GCSCR\_EL2().PCRSEL == '1';
        when EL3 => return GCSCR\_EL3().PCRSEL == '1';
    end;
    unreachable;
    return TRUE;
end;
```

Library pseudocode for aarch64/functions/gcs/GCSPOPCX

```
// GCSPOPCX()
// =====
// Called to pop and compare a GCS exception return record.

func GCSPOPCX()
begin
    let spsr : bits(64) = SPSR\_ELx();
    CheckGCSExRecord(ELR\_ELx(), spsr, X{64}(30), GCSInstType\_POPCX);
    PSTATE.EXLOCK = if GetCurrentEXLOCKEN() then '1' else '0';
    return;
end;
```

Library pseudocode for aarch64/functions/gcs/GCSPOPM

```
// GCSPOPM()
// =====
// Called to pop a GCS procedure return record.

func GCSPOPM() => bits(64)
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp\_LOAD, privileged);

    ptr = GetCurrentGCSPointer();
    let gcs_entry : bits(64) = Mem{64}(ptr, accdesc);

    if gcs_entry[1:0] != '00' then
        GCSDDataCheckException(GCSInstType\_POPM);
    end;

    ptr = ptr + 8;
    SetCurrentGCSPointer(ptr);
    return gcs_entry;
end;
```

Library pseudocode for aarch64/functions/gcs/GCSPOPX

```
// GCSPOPX()
// =====
// Called to pop a GCS exception return record.

func GCSPOPX()
begin
    var ptr : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp\_LOAD, privileged);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    let recorded_first_dword : bits(64) = Mem{64}(ptr, accdesc);
    if recorded_first_dword != Zeros{60}::'1001' then
        GCSDataCheckException(GCSInstType\_POPX);
    end;

    // Ignore these loaded values, however they might have
    // faulted which is why we load them anyway
    let recorded_elr : bits(64) = Mem{64}(ptr+8, accdesc);
    let recorded_spsr : bits(64) = Mem{64}(ptr+16, accdesc);
    let recorded_lr : bits(64) = Mem{64}(ptr+24, accdesc);

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPointer(ptr);
    return;
end;
```

Library pseudocode for aarch64/functions/gcs/GCSPUSHM

```
// GCSPUSHM()
// =====
// Called to push a GCS procedure return record.

func GCSPUSHM(value : bits(64))
begin
    AddGCSRecord(value);
    return;
end;
```

Library pseudocode for aarch64/functions/gcs/GCSPUSHX

```
// GCSPUSHX()
// =====
// Called to push a GCS exception return record.

func GCSPUSHX()
begin
    let spsr : bits(64) = SPSR\_ELx();
    AddGCSExRecord(ELR\_ELx(), spsr, X{64}(30));
    PSTATE.EXLOCK = '0';
    return;
end;
```

Library pseudocode for aarch64/functions/gcs/GCSReturnValueCheckEnabled

```
// GCSReturnValueCheckEnabled()
// =====
// Returns TRUE if the Guarded Control Stack has return value
// checking enabled at the current Exception level.

func GCSReturnValueCheckEnabled(el : bits(2)) => boolean
begin
    if UsingAArch32() then
        return FALSE;
    end;
    case el of
        when EL0 => return GCSCRE0\_EL1().RVCHKEN == '1';
        when EL1 => return GCSCR\_EL1().RVCHKEN == '1';
        when EL2 => return GCSCR\_EL2().RVCHKEN == '1';
        when EL3 => return GCSCR\_EL3().RVCHKEN == '1';
    end;
end;
```

Library pseudocode for aarch64/functions/gcs/GCSSS1

```
// GCSSS1()
// =====
// Operational pseudocode for GCSSS1 instruction.

func GCSSS1(incoming_pointer : bits(64))
begin
    var outgoing_pointer, cmpoperand, operand, data : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCSSS1(privileged);
    outgoing_pointer = GetCurrentGCSPointer();
    // Valid GCS cap record is expected
    cmpoperand = incoming_pointer[63:12]::'000000000001';
    // In-progress GCS cap record should be stored if the comparison is successful
    operand = outgoing_pointer[63:3]::'101';

    data = MemAtomic{64}(incoming_pointer, cmpoperand, operand, accdesc);
    if data == cmpoperand then
        SetCurrentGCSPointer(incoming_pointer[63:3]::'000');
    else
        GCSDataCheckException(GCSInstType\_SS1);
    end;
    return;
end;
```

Library pseudocode for aarch64/functions/gcs/GCSSS2

```
// GCSSS2()
// =====
// Operational pseudocode for GCSSS2 instruction.

func GCSSS2() => bits(64)
begin
    var outgoing_pointer, incoming_pointer, outgoing_value : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc_ld : AccessDescriptor = CreateAccDescGCS(MemOp\_LOAD, privileged);
    let accdesc_st : AccessDescriptor = CreateAccDescGCS(MemOp\_STORE, privileged);
    incoming_pointer = GetCurrentGCSPointer();
    outgoing_value = Mem{64}(incoming_pointer, accdesc_ld);

    if outgoing_value[2:0] == '101' then //in_progress token
        outgoing_pointer[63:3] = (outgoing_value[63:3]) - 1;
        outgoing_pointer[2:0] = '000';
        outgoing_value = outgoing_pointer[63:12]::'000000000001';
        Mem{64}(outgoing_pointer, accdesc_st) = outgoing_value;
        SetCurrentGCSPointer(incoming_pointer + 8);
        GCSSynchronizationBarrier();
    else
        GCSDataCheckException(GCSInstType\_SS2);
    end;
    return outgoing_pointer;
end;
```

Library pseudocode for aarch64/functions/gcs/GCSSTRapException

```
// GCSSTRapException()
// =====
// Handle a trap on GCSSTR instruction condition.

func GCSSTRapException(target_el : bits(2))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr()();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception\_GCSFail);
    except.syndrome.iss[24] = '0';
    except.syndrome.iss[23:20] = '0010';
    except.syndrome.iss[19:15] = '00000';
    except.syndrome.iss[14:10] = ThisInstr() [9:5];
    except.syndrome.iss[9:5] = ThisInstr() [4:0];
    except.syndrome.iss[4:0] = '00000';
    AArch64\_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/functions/gcs/GCSSynchronizationBarrier

```
// GCSSynchronizationBarrier()
// =====
// Barrier instruction that synchronizes GCS accesses
// in relation to other load and store accesses

impdef func GCSSynchronizationBarrier()
begin
    return;
end;
```

Library pseudocode for aarch64/functions/gcs/GetCurrentEXLOCKEN

```
// GetCurrentEXLOCKEN()
// =====

func GetCurrentEXLOCKEN() => boolean
begin
    if Halted\(\) || Restarting\(\) then
        return FALSE;
    end;

    case PSTATE.EL of
        when EL0 =>
            unreachable;
        when EL1 =>
            return GCSCR_EL1().EXLOCKEN == '1';
        when EL2 =>
            return GCSCR_EL2().EXLOCKEN == '1';
        when EL3 =>
            return GCSCR_EL3().EXLOCKEN == '1';
    end;
end;
```

Library pseudocode for aarch64/functions/gcs/GetCurrentGCSPtr

```
// GetCurrentGCSPtr()
// =====
// Returns the value of the current GCS pointer register.

func GetCurrentGCSPtr() => bits(64)
begin
    var ptr : bits(64);

    case PSTATE.EL of
        when EL0 =>
            ptr = GCSPR_EL0().PTR::'000';
        when EL1 =>
            ptr = GCSPR_EL1().PTR::'000';
        when EL2 =>
            ptr = GCSPR_EL2().PTR::'000';
        when EL3 =>
            ptr = GCSPR_EL3().PTR::'000';
    end;
    return ptr;
end;
```

Library pseudocode for aarch64/functions/gcs/LoadCheckGCSRecord

```
// LoadCheckGCSRecord()
// =====
// Validates the provided address against the top entry of the
// current Guarded Control Stack.

func LoadCheckGCSRecord(vaddress : bits(64), gcsinst_type : GCSInstruction) => bits(64)
begin
    var ptr : bits(64);
    var recorded_va : bits(64);
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescGCS(MemOp\_LOAD, privileged);

    ptr = GetCurrentGCSPtr();
    recorded_va = Mem{64}(ptr, accdesc);
    if GCSReturnValueCheckEnabled(PSTATE.EL) && (recorded_va != vaddress) then
        GCSDataCheckException(gcsinst_type);
    end;

    return recorded_va;
end;
```

Library pseudocode for aarch64/functions/gcs/SetCurrentGCSPtr

```
// SetCurrentGCSPtr()
// =====
// Writes a value to the current GCS ptr register.

func SetCurrentGCSPtr(ptr : bits(64))
begin
  case PSTATE.EL of
    when EL0 =>
      GCSPR_EL0().PTR = ptr[63:3];
    when EL1 =>
      GCSPR_EL1().PTR = ptr[63:3];
    when EL2 =>
      GCSPR_EL2().PTR = ptr[63:3];
    when EL3 =>
      GCSPR_EL3().PTR = ptr[63:3];
  end;
  return;
end;
```

Library pseudocode for aarch64/functions/hacdb

```
constant HACDB_ERR_REASON_STRUCTF : bits(2) = '01';
constant HACDB_ERR_REASON_IPAF : bits(2)    = '10';
constant HACDB_ERR_REASON_IPHACF : bits(2)  = '11';
```

Library pseudocode for aarch64/functions/hacdb/IsHACDBSIRQAsserted

```
// IsHACDBSIRQAsserted()
// =====
// Returns TRUE if HACDBSIRQ is asserted, and FALSE otherwise.

impdef func IsHACDBSIRQAsserted() => boolean
begin
  return FALSE;
end;
```



```

// ProcessHACDBSEntry()
// =====
// Process a single entry entry from the HACDBS.

func ProcessHACDBSEntry()
begin
    if !IsFeatureImplemented(FEAT_HACDBS) then return; end;

    if (HaveEL(EL3) && SCR_EL3().HACDBSEn == '0') || HACDBSBR_EL2().EN == '0' then
        SetInterruptRequestLevel(InterruptID HACDBSIRQ, LOW);
        return;
    end;

    if HCR_EL2().VM == '0' then return; end;

    if (UInt(HACDBSCONS_EL2().INDEX) >= (2 ^ (UInt(HACDBSBR_EL2().SZ) + 12)) DIVRM 8 ||
        HACDBSCONS_EL2().ERR_REASON != '00') then
        SetInterruptRequestLevel(InterruptID HACDBSIRQ, HIGH);
        return;
    elseif IsHACDBSIRQAsserted() then
        SetInterruptRequestLevel(InterruptID HACDBSIRQ, LOW);
    end;

    let hacdbbs_size : integer{} = UInt(HACDBSBR_EL2().SZ);
    var baddr : bits(NUM_PABITS) = HACDBSBR_EL2().BADDR[NUM_PABITS-13 : 0] :: Zeros{12};
    baddr[11 + hacdbbs_size : 12] = Zeros{hacdbbs_size};

    var accdesc : AccessDescriptor = CreateAccDescHACDBS();

    var addrdesc : AddressDescriptor;
    addrdesc.paddress.address = baddr + (8 * UInt(HACDBSCONS_EL2().INDEX));
    let nse2 : bit = '0'; // NSE2 has the Effective value of 0 within a PE.
    addrdesc.paddress.paspace = DecodePASpace(nse2, EffectiveSCR_EL3_NSE(), EffectiveSCR_EL3_NS());

    // Accesses to the HACDBS use the same memory attributes as used for stage 2 translation walks.
    addrdesc.memattrs = WalkMemAttrs(VTCR_EL2().SH0, VTCR_EL2().IRGN0, VTCR_EL2().ORGN0);
    let emec : bit = (if IsFeatureImplemented(FEAT_MEC) &&
        IsSCTLR2EL2Enabled() then SCTLR2_EL2().EMEC else '0');
    addrdesc.mecid = AArch64_S2TTWalkMECID(emec, accdesc.ss);

    var fault : FaultRecord = NoFault(accdesc);

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(addrdesc, accdesc);

        if fault.gpcf.gpf != GPCF_None then
            HACDBSCONS_EL2().ERR_REASON = HACDBS_ERR_REASON_STRUCTF;
            return;
        end;
    end;

    var memstatus : PhysMemRetStatus;
    var hacdbbs_entry : bits(64);
    (memstatus, hacdbbs_entry) = PhysMemRead{64}(addrdesc, accdesc);

    if IsFault(memstatus) then
        HACDBSCONS_EL2().ERR_REASON = HACDBS_ERR_REASON_STRUCTF;
        return;
    end;

    if BigEndian(accdesc.acctype) then
        hacdbbs_entry = BigEndianReverse{64}(hacdbbs_entry);
    end;

    // If the Valid field is clear, do not perform any cleaning operation
    // and increment HACDBSCONS_EL2().INDEX.
    if hacdbbs_entry[0] == '0' then
        HACDBSCONS_EL2().INDEX = HACDBSCONS_EL2().INDEX + 1;
        return;
    end;

    accdesc = CreateAccDescTTEUpdate(accdesc);
    var ipa : AddressDescriptor;
    ipa.paddress.address = hacdbbs_entry[NUM_PABITS-1:12] :: Zeros{12};

    let nsipa : bit = hacdbbs_entry[11];
    let paspace : PASpace = DecodePASpace(nse2, EffectiveSCR_EL3_NSE(), EffectiveSCR_EL3_NS());
    ipa.paddress.paspace = (if accdesc.ss == SS_Secure && nsipa == '1' then PAS_NonSecure
        else paspace);

    let slaarch64 : boolean = TRUE;
    let walkparams : S2TTWParams = AArch64_GetS2TTWParams(accdesc.ss, ipa.paddress.paspace,
        slaarch64);

    var descaddr : AddressDescriptor;
    var walkstate : TTWState;
    var descriptor : bits(128);
    if walkparams.d128 == '1' then
        (fault, descaddr, walkstate, descriptor) = AArch64_S2Walk{128}(fault, ipa, walkparams,

```

```

accdesc);
else
    (fault, descaddr, walkstate, descriptor[63:0]) = AArch64\_S2Walk{64}(fault, ipa, walkparams,
                                                                    accdesc);
end;

// If the Access flag on the Block or Page descriptor is set to 0, this does not generate
// an Access flag fault and the PE can still perform the cleaning operation on that descriptor.
if fault.statuscode == Fault\_AccessFlag then
    fault.statuscode = Fault\_None;
elseif fault.statuscode != Fault\_None then
    HACDBSCONS_EL2().ERR_REASON = HACDBS\_ERR\_REASON\_IPAF;
    return;
end;

let hacdbs_level : integer = SInt(hacdbs_entry[3:1]);
if walkstate.level != hacdbs_level || walkstate.contiguous == '1' then
    HACDBSCONS_EL2().ERR_REASON = HACDBS\_ERR\_REASON\_IPHACF;
    return;
end;

// For the purpose of cleaning HACDBS entries, it is not required that HW update of dirty bit
// is enabled for a descriptor to be qualified as writable-clean or writable-dirty.

// Check if the descriptor is neither writable-clean nor writable-dirty.
if walkparams.s2pie == '1' then
    let perms : S2AccessControls = AArch64\_S2ComputePermissions(walkstate.permissions,
                                                                walkparams, accdesc);

    if perms.w == '0' && perms.w.mmu == '0' then
        HACDBSCONS_EL2().ERR_REASON = HACDBS\_ERR\_REASON\_IPHACF;
        return;
    end;

// If DBM is 0, the descriptor is not writable-clean or writable-dirty.
elseif descriptor[51] == '0' then
    HACDBSCONS_EL2().ERR_REASON = HACDBS\_ERR\_REASON\_IPHACF;
    return;
end;

// If the descriptor is writable-clean, do not perform any cleaning
// operation and increment HACDBSCONS_EL2.INDEX.
if descriptor[7] == '0' then
    HACDBSCONS_EL2().INDEX = HACDBSCONS_EL2().INDEX + 1;
    return;
end;

var new_descriptor : bits(128) = descriptor;
new_descriptor[7] = '0';

let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);
if walkparams.d128 == '1' then
    (fault, -) = AArch64\_MemSwapTableDesc{128}(fault, descriptor, new_descriptor, walkparams.ee,
                                                descaccess, descaddr);
else
    (fault, -) = AArch64\_MemSwapTableDesc{64}(fault, descriptor[63:0], new_descriptor[63:0],
                                                walkparams.ee, descaccess, descaddr);
end;

if fault.statuscode != Fault\_None then
    HACDBSCONS_EL2().ERR_REASON = HACDBS\_ERR\_REASON\_IPAF;
else
    HACDBSCONS_EL2().INDEX = HACDBSCONS_EL2().INDEX + 1;
end;

return;
end;

```

Library pseudocode for aarch64/functions/ic/AArch64_CanTrapIC

```

// AArch64_CanTrapIC()
// =====
// Determines whether the execution of the IC instruction can be trapped.

func AArch64_CanTrapIC(cachetype : CacheType, cacheop : CacheOp, opscope : CacheOpScope) => boolean
begin
    return (!AArch64\_TreatICAsNOP(cachetype, cacheop, opscope) || ImpDefBool(
        "When IC is treated as NOP, data cache maintenance operations are trapped"));
end;

```


Library pseudocode for aarch64/functions/ic/AArch64_EffectiveOpScope

```
// AArch64_EffectiveOpScope()
// =====

func AArch64_EffectiveOpScope(opscope_in : CacheOpScope) => CacheOpScope
begin
    var opscope : CacheOpScope = opscope_in;

    if PSTATE.EL != EL1 then return opscope; end;

    if opscope == CacheOpScope\_ALLU && EL2Enabled() && HCR\_EL2().FB == '1' then
        opscope = CacheOpScope\_ALLUIS;
    end;

    return opscope;
end;
```

Library pseudocode for aarch64/functions/ic/AArch64_IC

```
// AArch64_IC()
// =====
// Perform Instruction Cache Operation.

func AArch64_IC(opscope : CacheOpScope)
begin
    let regval : bits(64) = ARBITRARY : bits(64);
    AArch64\_IC(regval, opscope);
end;

// AArch64_IC()
// =====
// Perform Instruction Cache Operation.

func AArch64_IC(regval : bits(64), opscope_in : CacheOpScope)
begin
    var cache : CacheRecord;
    var opscope : CacheOpScope = opscope_in;

    cache.acctype = AccessType\_IC;
    cache.cachetype = CacheType\_Instruction;
    cache.cacheop = CacheOp\_Invalidate;
    cache.opscope = opscope;

    if opscope IN {CacheOpScope\_ALLU,
                   CacheOpScope\_ALLUIS} then
        let ss : SecurityState = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        opscope = AArch64\_EffectiveOpScope(opscope);

        case opscope of
            when CacheOpScope\_ALLU =>
                cache.shareability = Shareability\_NSH;
            when CacheOpScope\_ALLUIS =>
                cache.shareability = Shareability\_ISH;
            otherwise =>
                unreachable;
        end;

        cache.regval = regval;
        CACHE\_OP(cache);
    else
        assert opscope == CacheOpScope\_PoU;

        if EL2Enabled() && !IsInHost() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid = VMID();
            else
                cache.is_vmid_valid = FALSE;
            end;
        else
            cache.is_vmid_valid = FALSE;
        end;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid = ASID();
        else
            cache.is_asid_valid = FALSE;
        end;

        let vaddress : bits(64) = regval;
        cache.vaddress = regval;

        let accdesc : AccessDescriptor = CreateAccDescIC(cache);
        let aligned : boolean = TRUE;
        let size : integer = 0;
        var memaddrdesc : AddressDescriptor = AArch64\_TranslateAddress(vaddress, accdesc,
                                                                    aligned, size);

        if IsFault(memaddrdesc) then
            memaddrdesc.fault.vaddress = regval;
            AArch64\_Abort(memaddrdesc.fault);
        end;

        cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
        cache.paddress = memaddrdesc.paddress;
        cache.shareability = memaddrdesc.memattr.shareability;
        if memaddrdesc.memattr.shareability == Shareability\_OSH then
            cache.shareability = Shareability\_ISH;
        end;

        CACHE\_OP(cache);
    end;
    return;
end;
```

Library pseudocode for aarch64/functions/ic/AArch64_TreatICAsNOP

```
// AArch64_TreatICAsNOP()
// =====
// Determines whether the execution of the IC instruction is treated as a NOP.

func AArch64_TreatICAsNOP(cachetype : CacheType, cacheop : CacheOp,
                           opscope : CacheOpScope) => boolean
begin
    if CTR_EL0().DIC == '1' then
        return ImpDefBool("IC is treated as NOP");
    end;

    return FALSE;
end;
```

Library pseudocode for aarch64/functions/immediateop/ImmediateOp

```
// ImmediateOp
// =====
// Vector logical immediate instruction types.

type ImmediateOp of enumeration {ImmediateOp_MOVI, ImmediateOp_MVNI,
                                 ImmediateOp_ORR, ImmediateOp_BIC};
```

Library pseudocode for aarch64/functions/logicalop/LogicalOp

```
// LogicalOp
// =====
// Logical instruction types.

type LogicalOp of enumeration {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

Library pseudocode for aarch64/functions/mec

```
constant DEFAULT_MECID : bits(16) = Zeros{};
```

Library pseudocode for aarch64/functions/mec/AArch64_S1AMECFault

```
// AArch64_S1AMECFault()
// =====
// Returns TRUE if a Translation fault should occur for Realm EL2 and Realm EL2&0
// stage 1 translated addresses to Realm PA space.

func AArch64_S1AMECFault{N}(walkparams : S1TTWParams, paspace : PASpace, regime : Regime,
                             descriptor : bits(N)) => boolean
begin
    assert N IN {64,128};
    let descriptor_amec : bit = (if walkparams.dl28 == '1' then descriptor[108]
                                else descriptor[63]);

    return (walkparams.[emec,amec] == '10' &&
            regime IN {Regime\_EL2, Regime\_EL20} &&
            paspace == PAS\_Realm &&
            descriptor_amec == '1');
end;
```

Library pseudocode for aarch64/functions/mec/AArch64_S1DisabledOutputMECID

```
// AArch64_S1DisabledOutputMECID()
// =====
// Returns the output MECID when stage 1 address translation is disabled.

func AArch64_S1DisabledOutputMECID(walkparams : S1TTWParams, regime : Regime,
                                   paspace : PASpace) => bits(16)
begin
    if walkparams.emec == '0' then
        return DEFAULT_MECID;
    end;

    if ! regime IN {Regime_EL2, Regime_EL20, Regime_EL10} then
        return DEFAULT_MECID;
    end;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;
    end;

    if regime == Regime_EL10 then
        return VMCID_P_EL2().MECID;
    else
        return MECID_P0_EL2().MECID;
    end;
end;
```

Library pseudocode for aarch64/functions/mec/AArch64_S1OutputMECID

```
// AArch64_S1OutputMECID()
// =====
// Returns the output MECID when stage 1 address translation is enabled.

func AArch64_S1OutputMECID{N}(walkparams : S1TTWParams, regime : Regime, varange : VARange,
                               paspace : PASpace, descriptor : bits(N)) => bits(16)
begin
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT_MECID;
    end;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;
    end;

    let descriptor_amec : bit = (if walkparams.dl28 == '1' then descriptor[108]
                                else descriptor[63]);

    case regime of
        when Regime_EL3 =>
            return MECID_RL_A_EL3().MECID;
        when Regime_EL2 =>
            if descriptor_amec == '0' then
                return MECID_P0_EL2().MECID;
            else
                return MECID_A0_EL2().MECID;
            end;
        when Regime_EL20 =>
            if varange == VARange_LOWER then
                if descriptor_amec == '0' then
                    return MECID_P0_EL2().MECID;
                else
                    return MECID_A0_EL2().MECID;
                end;
            else
                if descriptor_amec == '0' then
                    return MECID_P1_EL2().MECID;
                else
                    return MECID_A1_EL2().MECID;
                end;
            end;
        // Stage 2 translation might later override the MECID according to AMEC configuration.
        when Regime_EL10 =>
            return VMCID_P_EL2().MECID;
    end;
end;
```

Library pseudocode for aarch64/functions/mec/AArch64_S1TTWalkMECID

```
// AArch64_S1TTWalkMECID()
// =====
// Returns the associated MECID for the stage 1 translation table walk of the given
// translation regime and Security state.

func AArch64_S1TTWalkMECID(emec : bit, regime : Regime, ss : SecurityState) => bits(16)
begin
    if emec == '0' then
        return DEFAULT\_MECID;
    end;

    if ss != SS Realm then
        return DEFAULT\_MECID;
    end;

    case regime of
        when Regime\_EL2 =>
            return MECID_P0_EL2().MECID;
        when Regime\_EL20 =>
            if TCR_EL2().A1 == '0' then
                return MECID_P1_EL2().MECID;
            else
                return MECID_P0_EL2().MECID;
            end;
        // Stage 2 translation for a stage 1 walk might later override the
        // MECID according to AMEC configuration.
        when Regime\_EL10 =>
            return VMECID_P_EL2().MECID;
        otherwise =>
            unreachable;
    end;
end;
```

Library pseudocode for aarch64/functions/mec/AArch64_S2OutputMECID

```
// AArch64_S2OutputMECID()
// =====
// Returns the output MECID for stage 2 address translation.

func AArch64_S2OutputMECID{N}(walkparams : S2TTWParams, paspace : PASpace,
                               descriptor : bits(N)) => bits(16)
begin
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT\_MECID;
    end;

    if paspace != PAS Realm then
        return DEFAULT\_MECID;
    end;

    let descriptor_amec : bit = (if walkparams.d128 == '1' then descriptor[108]
                                else descriptor[63]);

    if descriptor_amec == '0' then
        return VMECID_P_EL2().MECID;
    else
        return VMECID_A_EL2().MECID;
    end;
end;
```

Library pseudocode for aarch64/functions/mec/AArch64_S2TTWalkMECID

```
// AArch64_S2TTWalkMECID()
// =====
// Returns the associated MECID for the stage 2 translation table walk of the
// given Security state.

func AArch64_S2TTWalkMECID(emec : bit, ss : SecurityState) => bits(16)
begin
    if emec == '0' then
        return DEFAULT\_MECID;
    end;

    if ss != SS Realm then
        return DEFAULT\_MECID;
    end;

    //Stage 2 translation might later override the MECID according to AMEC configuration
    return VMECID_P_EL2().MECID;
end;
```

Library pseudocode for aarch64/functions/memory/AArch64_AccessIsTagChecked

```
// AArch64_AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.

func AArch64_AccessIsTagChecked(vaddr : bits(64), accdesc : AccessDescriptor) => boolean
begin
    assert accdesc.tagchecked;

    if UsingAArch32() then
        return FALSE;
    end;

    if !IsMTEEnabled(accdesc.el) then
        return FALSE;
    end;

    if PSTATE.TCO == '1' then
        return FALSE;
    end;

    if (Halted() && EDSCR().MA == '1' &&
        ConstrainUnpredictableBool(Unpredictable NODTRTAGCHK)) then
        return FALSE;
    end;

    if (IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && !accdesc.write &&
        StoreOnlyTagCheckingEnabled(accdesc.el)) then
        return FALSE;
    end;

    let is_instr : boolean = FALSE;
    if (EffectiveMTX(vaddr, is_instr, PSTATE.EL) == '0' &&
        EffectiveTBI(vaddr, is_instr, PSTATE.EL) == '0') then
        return FALSE;
    end;

    if (EffectiveTCMA(vaddr, PSTATE.EL) == '1' &&
        (vaddr[59:55] == '00000' || vaddr[59:55] == '11111')) then
        return FALSE;
    end;

    return TRUE;
end;
```

Library pseudocode for aarch64/functions/memory/AArch64_AddressWithAllocationTag

```
// AArch64_AddressWithAllocationTag()
// =====
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.

func AArch64_AddressWithAllocationTag(address : bits(64), allocation_tag : bits(4)) => bits(64)
begin
    return address[63:60]::allocation_tag::address[55:0];
end;
```

Library pseudocode for aarch64/functions/memory/AArch64_AllocationTagCheck

```
// AArch64_AllocationTagCheck()
// =====
// Performs an Allocation Tag Check operation for a memory access and
// returns whether the check passed.

func AArch64_AllocationTagCheck(memaddrdesc : AddressDescriptor, accdesc : AccessDescriptor,
    ltag : bits(4)) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, memaddrdesc.vaddress);
    if memaddrdesc.memattrs.tags == MemTag\_AllocationTagged then
        var memstatus : PhysMemRetStatus;
        var readtag : bits(4);

        // Physical tagging needs no further translation, use the data PA to read the tag
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            let iswrite : boolean = FALSE;
            return ExternalFault(memstatus, iswrite, memaddrdesc, 1, accdesc);
        end;

        if ltag != readtag then
            fault.statuscode = Fault\_TagCheck;
        end;
    end;

    return fault;
end;
```

Library pseudocode for aarch64/functions/memory/AArch64_AllocationTagFromAddress

```
// AArch64_AllocationTagFromAddress()
// =====
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

func AArch64_AllocationTagFromAddress(tagged_address : bits(64)) => bits(4)
begin
    return tagged_address[59:56];
end;
```

Library pseudocode for aarch64/functions/memory/AArch64_CanonicalTagCheck

```
// AArch64_CanonicalTagCheck()
// =====
// Performs a Canonical Tag Check operation for a memory access and
// returns whether the check passed.

func AArch64_CanonicalTagCheck(memaddrdesc : AddressDescriptor, ltag : bits(4)) => boolean
begin
    let expected_tag : bits(4) = if memaddrdesc.vaddress[55] == '0' then '0000' else '1111';
    return ltag == expected_tag;
end;
```

Library pseudocode for aarch64/functions/memory/AArch64_CheckTag

```
// AArch64_CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns a FaultRecord indicating if
// the check passed. If Tag Check Faults are asynchronously accumulated, a Tag Check Fault
// exception is recorded in TFSR_ELx.

func AArch64_CheckTag(memaddrdesc_in : AddressDescriptor, accdesc : AccessDescriptor,
    readcheck : boolean, size : integer, ltag : bits(4)) => FaultRecord
begin
    var memaddrdesc : AddressDescriptor = memaddrdesc_in;
    // NoFault() will set fault.write to FALSE to accesses that perform both a read and a write.
    var fault : FaultRecord = NoFault(accdesc, memaddrdesc.vaddress);
    let granules : integer = Max(size DIVRM TAG GRANULE, 1);
    let forcesync : boolean = accdesc.nonfault || (accdesc.firstfault && !accdesc.first);
    let tcf : TCFType = AArch64\_EffectiveTCF(accdesc.el, readcheck);

    for i = 0 to granules - 1 do
        case memaddrdesc.memattrs.tags of
            when MemTag\_AllocationTagged =>
                fault = AArch64\_AllocationTagCheck(memaddrdesc, accdesc, ltag);
                if fault.statuscode == Fault\_TagCheck then
                    if tcf == TCFType\_Sync || forcesync then
                        fault.statuscode = Fault\_TagCheck;
                        return fault;
                    elsif tcf == TCFType\_Async then
                        AArch64\_ReportTagCheckFault(accdesc.el, memaddrdesc.vaddress[55]);
                        fault.statuscode = Fault\_None;
                        return fault;
                    else
                        // Tag Check Faults have no effect on the PE.
                        fault.statuscode = Fault\_None;
                    end;
                elsif fault.statuscode != Fault\_None then
                    return fault;
                end;
            when MemTag\_CanonicallyTagged =>
                if !AArch64\_CanonicalTagCheck(memaddrdesc, ltag) then
                    if tcf == TCFType\_Sync || forcesync then
                        fault.statuscode = Fault\_TagCheck;
                        return fault;
                    elsif tcf == TCFType\_Async then
                        AArch64\_ReportTagCheckFault(accdesc.el, memaddrdesc.vaddress[55]);
                        return fault;
                    else
                        // Tag Check Faults have no effect on the PE.
                        pass;
                    end;
                end;
            when MemTag\_Untagged =>
                pass;
            otherwise =>
                unreachable;
        end;

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + TAG GRANULE;
        memaddrdesc.vaddress = memaddrdesc.vaddress + TAG GRANULE;
    end;

    return fault;
end;
```

Library pseudocode for aarch64/functions/memory/AArch64_IsUnprivAccessPriv

```
// AArch64_IsUnprivAccessPriv()
// =====
// Returns TRUE if an unprivileged access is privileged, and FALSE otherwise.

func AArch64_IsUnprivAccessPriv() => boolean
begin
    var privileged : boolean;

    case PSTATE.EL of
        when EL0 =>
            privileged = FALSE;
        when EL1 => privileged = EffectiveHCR_EL2_NVx()[1:0] == '11';
        when EL2 => privileged = !ELIsInHost(EL0);
        when EL3 =>
            privileged = TRUE;
    end;

    if IsFeatureImplemented(FEAT_UAO) && PSTATE.UAO == '1' then
        privileged = PSTATE.EL != EL0;
    end;

    return privileged;
end;
```

Library pseudocode for aarch64/functions/memory/AArch64_LogicalAddressTag

```
// AArch64_LogicalAddressTag()
// =====
// Extract the Logical Address Tag from an address

func AArch64_LogicalAddressTag(vaddr : bits(64)) => bits(4)
begin
    return vaddr[59:56];
end;
```


Library pseudocode for aarch64/functions/memory/AArch64_MemSingle

```
// AArch64_MemSingle - accessor
// =====

accessor AArch64_MemSingle{size : integer{8, 16, 32, 64, 128, 256}}(address : bits(64),
                                                                    accdesc : AccessDescriptor,
                                                                    aligned : boolean
                                                                    ) <=> value : bits(size)

begin
    // Perform an atomic, little-endian read of 'size' bits.
    getter
        let bytes : integer{} = size DIV 8;
        var value : bits(size);
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        (value, memaddrdesc, memstatus) = AArch64\_MemSingleRead{size}(address, accdesc, aligned);

        // Check for a fault from translation or the output of translation.
        if IsFault(memaddrdesc) then
            AArch64\_Abort(memaddrdesc.fault);
        end;

        // Check for external aborts.
        if IsFault(memstatus) then
            HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, bytes, accdesc);
        end;

        return value;
    end;

    // Perform an atomic, little-endian write of 'size' bits.
    setter
        let bytes : integer{} = size DIV 8;
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        (memaddrdesc, memstatus) = AArch64\_MemSingleWrite{size}(address, accdesc, aligned, value);

        // Check for a fault from translation or the output of translation.
        if IsFault(memaddrdesc) then
            AArch64\_Abort(memaddrdesc.fault);
        end;

        // Check for external aborts.
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;
    end;
end;
```



```

// AArch64_MemSingleRead()
// =====
// Perform an atomic, little-endian read of 'size' bits.

func AArch64_MemSingleRead(size : integer{8, 16, 32, 64, 128, 256})(address : bits(64),
                                                                    accdesc in : AccessDescriptor,
                                                                    aligned : boolean
                                                                    ) => (bits(size),
                                                                    AccessDescriptor,
                                                                    PhysMemRetStatus)

begin
    var value : bits(size) = ARBITRARY : bits(size);
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;
    let bytes : integer{} = size DIV 8;
    memstatus.statuscode = Fault\_None;
    var accdesc : AccessDescriptor = accdesc_in;
    if IsFeatureImplemented(FEAT_LSE2) then
        let quantity : integer = MemSingleGranule();
        assert ((IsFeatureImplemented(FEAT_LS64WB) &&
                bytes == 32 && accdesc.acctype == AccessType\_ASIMD) ||
                AllInAlignedQuantity{64}(address, bytes, quantity));
    else
        assert IsAlignedSize(address, bytes);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64\_AccessIsTagChecked(address, accdesc);
    end;

    var memaddrdesc : AccessDescriptor;
    memaddrdesc = AArch64\_TranslateAddress(address, accdesc, aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (value, memaddrdesc, memstatus);
    end;

    // Memory array access
    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64\_LogicalAddressTag(address);
        let readcheck : boolean = TRUE;
        let fault : FaultRecord = AArch64\_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault\_None then
            memaddrdesc.fault = fault;
            return (value, memaddrdesc, memstatus);
        end;
    end;

    if accdesc.acctype != AccessType\_IFETCH && SPESampleInFlight then
        let is_load : boolean = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;
    var atomic : boolean;
    if IsWBShareable(memaddrdesc.memattrs) then
        atomic = TRUE;
    elsif accdesc.exclusive then
        atomic = TRUE;
    elsif (accdesc.acctype == AccessType\_SVE && accdesc.predicated &&
            bytes == 8 && IsAlignedSize(address, 8)) then
        // An SVE predicated load of a 128-bit element that is 64-bit aligned
        // is treated as a pair of 64-bit single-copy atomic accesses.
        // This is one of the 64-bit single-copy atomic access.
        atomic = TRUE;
    elsif aligned then
        atomic = !accdesc.ispair;
    else
        // Misaligned accesses within MemSingleGranule() byte aligned memory but
        // not Normal Cacheable Writeback are Atomic
        atomic = ImpDefBool("FEAT_LSE2: access is atomic");
    end;

    if atomic then
        (memstatus, value) = PhysMemRead{size}(memaddrdesc, accdesc);
    elsif accdesc.acctype == AccessType\_ASIMD && bytes == 32 && accdesc.ispair then
        // A 32 byte LDP (SIMD&FP) that does not target Normal Inner Write-Back, Outer
        // Write-Back cacheable, Shareable memory is treated as four 8 byte atomic accesses.
        // As this access was not split in Mem(), it must be aligned to 32 bytes.
        assert IsAlignedSize(address, 32);
        accdesc.ispair = FALSE;
        for i = 0 to 3 do
            if IsFault(memstatus) then // Do not continue past a fault
                (memstatus, value[i*64+:64]) = PhysMemRead{64}(memaddrdesc, accdesc);
            end;
        end;

        if IsFault(memstatus) then
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 8;
        end;
    end;
end

```

```

        memaddrdesc.vaddress      = memaddrdesc.vaddress + 8;
    end;
end;

elsif aligned && accdesc.ispair then
    let half : integer{} = (size DIV 2) as integer{32, 64, 128};
    (memstatus, value[0*:half]) = PhysMemRead(half)(memaddrdesc, accdesc);

    if !IsFault(memstatus) then
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + (half DIV 8);
        memaddrdesc.vaddress        = memaddrdesc.vaddress + (half DIV 8);
        (memstatus, value[1*:half]) = PhysMemRead(half)(memaddrdesc, accdesc);
    end;

else
    for i = 0 to bytes-1 do
        if !IsFault(memstatus) then          // Do not continue past a fault
            (memstatus, value[i*:8]) = PhysMemRead(8)(memaddrdesc, accdesc);
        end;

        if !IsFault(memstatus) then
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress        = memaddrdesc.vaddress + 1;
        end;
    end;
end;

if IsFault(memstatus) then
    return (value, memaddrdesc, memstatus);
end;

if accdesc.acctype == AccessType\_IFETCH then
    memaddrdesc.fault = AArch64\_CheckDebug(address, accdesc, bytes);
end;

return (value, memaddrdesc, memstatus);
end;

```



```

// AArch64_MemSingleWrite()
// =====
// Perform an atomic, little-endian write of 'size' bits.

func AArch64_MemSingleWrite{size : integer{8, 16, 32, 64, 128, 256}}(address : bits(64),
                                                                    accdesc_in : AccessDescriptor,
                                                                    aligned : boolean,
                                                                    value : bits(size)
                                                                    ) => (AddressDescriptor, PhysMemRetStatus)

begin
    var accdesc : AccessDescriptor = accdesc_in;
    let bytes : integer{} = size DIV 8;
    if IsFeatureImplemented(FEAT_LSE2) then
        let quantity : integer = MemSingleGranule();
        assert ((IsFeatureImplemented(FEAT_LS64WB) &&
                bytes == 32 && accdesc.acctype == AccessType\_ASIMD) ||
                AllInAlignedQuantity{64}(address, bytes, quantity));
    else
        assert IsAlignedSize(address, bytes);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64\_AccessIsTagChecked(address, accdesc);
    end;

    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;
    memaddrdesc = AArch64\_TranslateAddress(address, accdesc, aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (memaddrdesc, memstatus);
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64\_LogicalAddressTag(address);
        let readcheck : boolean = FALSE;
        let fault : FaultRecord = AArch64\_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault\_None then
            memaddrdesc.fault = fault;
            return (memaddrdesc, memstatus);
        end;
    end;

    if SPESampleInFlight then
        let is_load : boolean = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;

    var atomic : boolean;
    if IsWBShareable(memaddrdesc.memattrs) then
        atomic = TRUE;
    elsif accdesc.exclusive then
        atomic = TRUE;
    elsif (accdesc.acctype == AccessType\_SVE && accdesc.predicated &&
            bytes == 8 && IsAlignedSize(address, 8)) then
        // An SVE predicated load of a 128-bit element that is 64-bit aligned
        // is treated as a pair of 64-bit single-copy atomic accesses.
        // This is one of the 64-bit single-copy atomic access.
        atomic = TRUE;
    elsif aligned then
        atomic = !accdesc.ispair;
    else
        // Misaligned accesses within MemSingleGranule() byte aligned memory but
        // not Normal Cacheable Writeback are Atomic
        atomic = ImpDefBool("FEAT_LSE2: access is atomic");
    end;

    if atomic then
        memstatus = PhysMemWrite(size)(memaddrdesc, accdesc, value);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);
        end;
    end;

    elsif accdesc.acctype == AccessType\_ASIMD && bytes == 32 && accdesc.ispair then
        // A 32 byte STP (SIMD&FP) that does not target Normal Inner Write-Back, Outer
        // Write-Back cacheable, Shareable memory is treated as four 8 byte atomic accesses.
        // As this access was not split in Mem(), it must be aligned to 32 bytes.
        assert IsAlignedSize(address, 32);
        accdesc.ispair = FALSE;
        for i = 0 to 3 do
            memstatus = PhysMemWrite{64}(memaddrdesc, accdesc, value[64*i+64]);
            if IsFault(memstatus) then

```

```

        return (memaddrdesc, memstatus);
    end;
    memaddrdesc.paddress.address = memaddrdesc.paddress.address + 8;
    memaddrdesc.vaddress        = memaddrdesc.vaddress + 8;
end;

elsif aligned && accdesc.ispair then
    let half : integer{} = (size DIV 2) as integer{32, 64};
    memstatus = PhysMemWrite(half)(memaddrdesc, accdesc, value[0*:half]);
    if IsFault(memstatus) then
        return (memaddrdesc, memstatus);
    end;

    memaddrdesc.paddress.address = memaddrdesc.paddress.address + (half DIV 8);
    memaddrdesc.vaddress        = memaddrdesc.vaddress + (half DIV 8);
    memstatus = PhysMemWrite(half)(memaddrdesc, accdesc, value[1*:half]);
    if IsFault(memstatus) then
        return (memaddrdesc, memstatus);
    end;

else
    for i = 0 to bytes-1 do
        memstatus = PhysMemWrite(8)(memaddrdesc, accdesc, value[i*:8]);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);
        end;
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
        memaddrdesc.vaddress        = memaddrdesc.vaddress + 1;
    end;
end;

return (memaddrdesc, memstatus);
end;

```

Library pseudocode for aarch64/functions/memory/AArch64_MemTag

```

// AArch64_MemTag - accessor
// =====

accessor AArch64_MemTag(address : bits(64), accdesc : AccessDescriptor) <=> value : bits(4)
begin
    // Load an Allocation Tag from memory.
    getter
        var value : bits(4);
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;
        (value, memaddrdesc, memstatus) = AArch64\_MemTagRead(address, accdesc);

        if IsFault(memaddrdesc) then
            AArch64\_Abort(memaddrdesc.fault);
        end;

        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        end;

        return value;
    end;

    // Store an Allocation Tag to memory.
    setter
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;
        (memaddrdesc, memstatus) = AArch64\_MemTagWrite(address, accdesc, value);

        if IsFault(memaddrdesc) then
            AArch64\_Abort(memaddrdesc.fault);
        end;

        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        end;
    end;
end;
end;

```

Library pseudocode for aarch64/functions/memory/AArch64_MemTagRead

```
// AArch64_MemTagRead()
// =====
// Load an Allocation Tag from memory.

func AArch64_MemTagRead(address : bits(64),
    accdesc_in : AccessDescriptor
    ) => (bits(4), AddressDescriptor, PhysMemRetStatus)

begin
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    var accdesc : AccessDescriptor = accdesc_in;
    var tag : bits(4) = ARBITRARY : bits(4);
    var memaddrdesc : AddressDescriptor = ARBITRARY : AddressDescriptor;
    var memtagtype : MemTagType = ARBITRARY : MemTagType;
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;

    let aligned : boolean = TRUE;

    accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    (memtagtype, memaddrdesc) = AArch64\_TranslateTagAddress(address, accdesc, aligned, TAG\_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (tag, memaddrdesc, memstatus);
    end;

    if accdesc.tagaccess && memtagtype == MemTag\_AllocationTagged then
        (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
        return (tag, memaddrdesc, memstatus);
    elsif (IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) &&
        accdesc.tagaccess && memtagtype == MemTag\_CanonicallyTagged) then
        tag = if address[55] == '0' then '0000' else '1111';
        return (tag, memaddrdesc, memstatus);
    else
        // Otherwise read the tag as zero
        tag = '0000';
        return (tag, memaddrdesc, memstatus);
    end;
end;
```

Library pseudocode for aarch64/functions/memory/AArch64_MemTagWrite

```
// AArch64_MemTagWrite()
// =====
// Store an Allocation Tag to memory.

func AArch64_MemTagWrite(address : bits(64),
    accdesc_in : AccessDescriptor,
    value : bits(4)) => (AddressDescriptor, PhysMemRetStatus)

begin
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    var accdesc : AccessDescriptor = accdesc_in;
    var memaddrdesc : AddressDescriptor = ARBITRARY : AddressDescriptor;
    var memtagtype : MemTagType = ARBITRARY : MemTagType;
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;

    let aligned : boolean = IsAlignedSize(address, TAG\_GRANULE);

    // Stores of allocation tags must be aligned
    if !aligned then
        memaddrdesc.fault = AlignmentFault(accdesc, address);
        return (memaddrdesc, memstatus);
    end;

    accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    (memtagtype, memaddrdesc) = AArch64\_TranslateTagAddress(address, accdesc, aligned, TAG\_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (memaddrdesc, memstatus);
    end;

    if accdesc.tagaccess && memtagtype == MemTag\_AllocationTagged then
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
    end;
    return (memaddrdesc, memstatus);
end;
```


Library pseudocode for aarch64/functions/memory/AArch64_UnalignedAccessFaults

```
// AArch64_UnalignedAccessFaults()
// =====
// Determine whether the unaligned access generates an Alignment fault

func AArch64_UnalignedAccessFaults(accdesc : AccessDescriptor,
                                   address : bits(64),
                                   size : integer) => boolean

begin
  if AlignmentEnforced() then
    return TRUE;
  elsif accdesc.acctype == AccessType\_GCS then
    return TRUE;
  elsif accdesc.rcw then
    return TRUE;
  elsif accdesc.ls64 then
    return TRUE;
  elsif (accdesc.exclusive || accdesc.atomicop) then
    let quantity : integer = MemSingleGranule();
    return (!IsFeatureImplemented(FEAT_LSE2) ||
            !AllInAlignedQuantity{64}(address, size, quantity));
  elsif (accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
    // If nAA is 0, the address accessed by each register in the pair
    // must lie within a single 16-byte aligned block
    if accdesc.ispair then
      return (SCTLR\_ELx().nAA == '0' &&
              (!AllInAlignedQuantity{64}(address, size DIV 2, 16) ||
               !AllInAlignedQuantity{64}(address + size DIV 2, size DIV 2, 16)));
    else
      return (SCTLR\_ELx().nAA == '0' &&
              !AllInAlignedQuantity{64}(address, size, 16));
    end;
  else
    return FALSE;
  end;
end;
```

Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

```
// AddressSupportsLS64()
// =====
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.

impdef func AddressSupportsLS64(paddress : bits(NUM\_PABITS)) => boolean
begin
  return ImpDefBool("Memory location supports ST64B and LD64B");
end;
```

Library pseudocode for aarch64/functions/memory/CASCompare

```
// CASCompare()
// =====
// Performs a comparison for CAS

func CASCompare{N}(oldvalue : bits(N),
                   comparevalue : bits(N),
                   newvalue : bits(N)) => (bits(N), boolean, bits(N))

begin
  var memresult : bits(N);
  var cmpfail : boolean;
  var regresult : bits(N) = oldvalue;

  if oldvalue == comparevalue then
    cmpfail = FALSE;
    memresult = newvalue;
    if !ConstrainUnpredictableBool(Unpredictable\_CASRETURNOLDVALUE) then
      regresult = comparevalue;
    end;
  else
    cmpfail = TRUE;
    memresult = oldvalue;
  end;

  return (memresult, cmpfail, regresult);
end;
```

Library pseudocode for aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state.

func CheckSPAlignment()
begin
    let sp : bits(64) = SP{}();
    var stack_align_check : boolean;
    if PSTATE.EL == ELO then
        stack_align_check = (SCTLR_ELx().SA0 != '0');
    else
        stack_align_check = (SCTLR_ELx().SA != '0');
    end;

    if stack_align_check && sp != AlignDownSize(sp, 16) then
        AArch64_SPAlignmentFault();
    end;

    return;
end;
```

Library pseudocode for aarch64/functions/memory/IsConventionalMemory

```
// IsConventionalMemory()
// =====
// Returns TRUE if the memory location is in Conventional memory, and FALSE otherwise.

impdef func IsConventionalMemory(addrdesc : AddressDescriptor) => boolean
begin
    return TRUE;
end;
```



```

// Mem - accessor
// =====

accessor Mem{size : integer{8, 16, 32, 64, 128, 256}}(address : bits(64),
                                                    accdesc_in : AccessDescriptor
                                                    ) <=> value_in : bits(size)

begin
    // Perform a read of 'size' bits. The access byte order is reversed for a big-endian access.
    // Instruction fetches would call AArch64_MemSingle directly.
    getter
        var accdesc : AccessDescriptor = accdesc_in;
        let bytes : integer{} = size DIV 8;
        var value : bits(size);

        // Check alignment on size of element accessed, not overall access size
        var aligned : boolean;
        if accdesc.ispair && !accdesc.exclusive then
            let half : integer{} = (size DIV 2) as integer{32, 64, 128};
            aligned = IsAlignedSize(address, half DIV 8);
        else
            aligned = IsAlignedSize(address, bytes);
        end;

        let quantity : integer = MemSingleGranule();

        if !aligned && AArch64 UnalignedAccessFaults(accdesc, address, bytes) then
            let fault : FaultRecord = AlignmentFault(accdesc, address);
            AArch64 Abort(fault);
        end;

        if accdesc.acctype == AccessType ASIMD && bytes == 16 && IsAlignedSize(address, 8) then
            // If 128-bit SIMD&FP ordered access are treated as a pair of
            // 64-bit single-copy atomic accesses, then these single copy atomic
            // access can be observed in any order.
            let half : integer{} = (size DIV 2) as integer{64};
            let highaddress : bits(64) = AddressIncrement(address, half DIV 8, accdesc);
            value[0*:half] = AArch64 MemSingle{half}(address, accdesc, aligned);
            value[1*:half] = AArch64 MemSingle{half}(highaddress, accdesc, aligned);
        elsif (IsFeatureImplemented(FEAT_LS64WB) && accdesc.acctype == AccessType ASIMD &&
            bytes == 32 && accdesc.ispair && IsAlignedSize(address, 32)) then
            value = AArch64 MemSingle{size}(address, accdesc, aligned);
        elsif accdesc.acctype == AccessType ASIMD && bytes == 32 && IsAlignedSize(address, 8) then
            // If a 32 byte LDP (SIMD&FP) access is not aligned to 32 bytes but aligned to
            // 8 bytes, it is treated as four 8 byte single-copy atomic accesses.
            accdesc.ispair = FALSE;
            aligned = TRUE;
            for i = 0 to 3 do
                let blockaddress : bits(64) = AddressIncrement(address, i*8, accdesc);
                value[64*i+:64] = AArch64 MemSingle{64}(blockaddress, accdesc, aligned);
            end;
        elsif (IsFeatureImplemented(FEAT_LSE2) &&
            AllInAlignedQuantity{64}(address, bytes, quantity)) then
            value = AArch64 MemSingle{size}(address, accdesc, aligned);
        elsif ((aligned && accdesc.ispair) ||
            (accdesc.acctype == AccessType SVE && accdesc.predicated &&
            bytes == 16 && IsAlignedSize(address, 8))) then
            // Either: an aligned pair access, OR
            // an SVE predicated load of a 128-bit element that is 64-bit aligned,
            // which is treated as two 64-bit single-copy atomic accesses.
            accdesc.ispair = FALSE;
            let half : integer{} = (size DIV 2) as integer{32, 64, 128};
            let highaddress : bits(64) = AddressIncrement(address, half DIV 8, accdesc);
            if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.highestaddressfirst then
                value[1*:half] = AArch64 MemSingle{half}(highaddress, accdesc, aligned);
                value[0*:half] = AArch64 MemSingle{half}(address, accdesc, aligned);
            else
                value[0*:half] = AArch64 MemSingle{half}(address, accdesc, aligned);
                value[1*:half] = AArch64 MemSingle{half}(highaddress, accdesc, aligned);
            end;
        elsif aligned then
            value = AArch64 MemSingle{size}(address, accdesc, aligned);
        else
            assert bytes > 1;
            if (IsFeatureImplemented(FEAT_LRCPC3) && accdesc.ispair &&
                accdesc.highestaddressfirst) then
                let half : integer{} = (size DIV 2) as integer{32, 64, 128};
                var lowhalf, highhalf : bits(half);

                for i = 0 to (half DIV 8)-1 do
                    let byteaddress : bits(64) = AddressIncrement(address,
                                                                (half DIV 8) + i, accdesc);
                    // Individual byte access can be observed in any order
                    highhalf[i*:8] = AArch64 MemSingle{8}(byteaddress, accdesc, aligned);
                end;
                for i = 0 to (half DIV 8)-1 do
                    let byteaddress : bits(64) = AddressIncrement(address, i, accdesc);
                    // Individual byte access can be observed in any order
                    lowhalf[i*:8] = AArch64 MemSingle{8}(byteaddress, accdesc, aligned);
                end;
            end;
        end;
    end;
end;

```

```

        value = highhalf::lowhalf;

    else
        value[7:0] = AArch64 MemSingle{8}(address, accdesc, aligned);
        accdesc.lowestaddress = FALSE;

        // For subsequent bytes, if they cross to a new translation page which assigns
        // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
        // will generate an Alignment Fault.
        let c : Constraint = ConstrainUnpredictable(Unpredictable DEVPAGE2);
        assert c IN {Constraint FAULT, Constraint NONE};
        if c == Constraint NONE then aligned = TRUE; end;

        for i = 1 to bytes-1 do
            let byteaddress : bits(64) = AddressIncrement(address, i, accdesc);
            value[i*8] = AArch64 MemSingle{8}(byteaddress, accdesc, aligned);
        end;
    end;
end;

if BigEndian(accdesc.acctype) then
    value = BigEndianReverse(size)(value);
end;

return value;
end;

// Perform a write of 'size' bits. The byte order is reversed for a big-endian access.
setter
var value : bits(size) = value_in;
let bytes : integer{} = size DIV 8;
var accdesc : AccessDescriptor = accdesc_in;

// Check alignment on size of element accessed, not overall access size
var aligned : boolean;
if accdesc.ispair && !accdesc.exclusive then
    let half : integer{} = (size DIV 2) as integer{32, 64, 128};
    aligned = IsAlignedSize(address, half DIV 8);
else
    aligned = IsAlignedSize(address, bytes);
end;
let quantity : integer = MemSingleGranule();

if !aligned && AArch64 UnalignedAccessFaults(accdesc, address, bytes) then
    let fault : FaultRecord = AlignmentFault(accdesc, address);
    AArch64 Abort(fault);
end;

if BigEndian(accdesc.acctype) then
    value = BigEndianReverse(size)(value);
end;
if accdesc.acctype == AccessType ASIMD && bytes == 16 && IsAlignedSize(address, 8) then
    let half : integer{} = (size DIV 2) as integer{64};
    // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
    // 64-bit aligned.
    let highaddress : bits(64) = AddressIncrement(address, half DIV 8, accdesc);
    AArch64 MemSingle{half}(address, accdesc, aligned) = value[0+:half];
    AArch64 MemSingle{half}(highaddress, accdesc, aligned) = value[half+:half];
elseif IsFeatureImplemented(FEAT_LS64WB) && accdesc.acctype == AccessType ASIMD &&
    bytes == 32 && accdesc.ispair && IsAlignedSize(address, 32) then
    AArch64 MemSingle(size)(address, accdesc, aligned) = value;
elseif accdesc.acctype == AccessType ASIMD && bytes == 32 && IsAlignedSize(address, 8) then
    // If a 32 byte STP (SIMD&FP) access is not aligned to 32 bytes but aligned to
    // 8 bytes, it is treated as four 8 byte single-copy atomic accesses.
    accdesc.ispair = FALSE;
    aligned = TRUE;
    for i = 0 to 3 do
        let blockaddress : bits(64) = AddressIncrement(address, i*8, accdesc);
        AArch64 MemSingle{64}(blockaddress, accdesc, aligned) = value[64*i+:64];
    end;
elseif IsFeatureImplemented(FEAT_LSE2) &&
    AllInAlignedQuantity{64}(address, bytes, quantity) then
    AArch64 MemSingle(size)(address, accdesc, aligned) = value;
elseif ((aligned && accdesc.ispair) ||
    (accdesc.acctype == AccessType SVE && accdesc.predicated &&
    bytes == 16 && IsAlignedSize(address, 8))) then
    // Either: an aligned pair access, OR
    // an SVE predicated load of a 128-bit element that is 64-bit aligned,
    // which is treated as two 64-bit single-copy atomic accesses.
    let half : integer{} = (size DIV 2) as integer{32, 64, 128};
    accdesc.ispair = FALSE;
    let highaddress : bits(64) = AddressIncrement(address, half DIV 8, accdesc);
    if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.highestaddressfirst then
        AArch64 MemSingle{half}(highaddress, accdesc, aligned) = value[half+:half];
        AArch64 MemSingle{half}(address, accdesc, aligned) = value[0+:half];
    else
        AArch64 MemSingle{half}(address, accdesc, aligned) = value[0+:half];
        AArch64 MemSingle{half}(highaddress, accdesc, aligned) = value[half+:half];
    end;
end;

```

```

elseif aligned then
    AArch64 MemSingle{size}(address, accdesc, aligned) = value;
else
    assert bytes > 1;
    if (IsFeatureImplemented(FEAT_LRCPC3) && accdesc.ispair &&
        accdesc.highestaddressfirst) then
        let half : integer{} = (size DIV 2) as integer{32, 64};
        var lowhalf, highhalf : bits(half);
        (highhalf, lowhalf) = (value[half+:half], value[0+:half]);
        for i = 0 to (half DIV 8)-1 do
            let byteaddress : bits(64) = AddressIncrement(address,
                (half DIV 8) + i, accdesc);
            // Individual byte access can be observed in any order
            AArch64 MemSingle{8}(byteaddress, accdesc, aligned) = highhalf[i*:8];
        end;
        for i = 0 to (half DIV 8)-1 do
            let byteaddress : bits(64) = AddressIncrement(address,
                (half DIV 8) + i, accdesc);
            // Individual byte access can be observed in any order,
            // but implies observability of highhalf
            AArch64 MemSingle{8}(byteaddress, accdesc, aligned) = lowhalf[i*:8];
        end;
    else
        AArch64 MemSingle{8}(address, accdesc, aligned) = value[7:0];
        accdesc.lowestaddress = FALSE;

        // For subsequent bytes, if they cross to a new translation page which assigns
        // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
        // will generate an Alignment Fault.

        let c : Constraint = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
        assert c IN {Constraint\_FAULT, Constraint\_NONE};
        if c == Constraint\_NONE then aligned = TRUE; end;

        for i = 1 to bytes-1 do
            let byteaddress : bits(64) = AddressIncrement(address, i, accdesc);
            AArch64 MemSingle{8}(byteaddress, accdesc, aligned) = value[i*:8];
        end;
    end;
end;
end;
end;
end;
end;

```



```

// MemAtomic()
// =====
// Performs load and store memory operations for a given virtual address.

func MemAtomic{size : integer{8, 16, 32, 64, 128}}(address : bits(64),
                                                    cmpoperand : bits(size),
                                                    operand : bits(size),
                                                    accdesc_in : AccessDescriptor
                                                    ) => bits(size)

begin
    assert accdesc_in.atomicop;

    let bytes : integer{} = (size DIV 8) as integer{1, 2, 4, 8, 16};

    var newvalue : bits(size);
    var oldvalue : bits(size);
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64 AccessIsTagChecked(address, accdesc);
    end;

    if !aligned && AArch64 UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64 Abort(fault);
    end;

    // MMU or MPU lookup
    let memaddrdesc : AddressDescriptor = AArch64 TranslateAddress(address, accdesc,
                                                                aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64 Abort(memaddrdesc.fault);
    end;

    if (!IsWBShareable(memaddrdesc.memattrs) &&
        ConstrainUnpredictableBool(Unpredictable Atomic NOP)) then
        return ARBITRARY : bits(size);
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
    end;

    // For Store-only Tag checking, the tag check is performed on the store.
    if (accdesc.tagchecked &&
        (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
         !StoreOnlyTagCheckingEnabled(accdesc.el))) then
        let ltag : bits(4) = AArch64 LogicalAddressTag(address);
        let readcheck : boolean = TRUE;
        let fault : FaultRecord = AArch64 CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault None then
            AArch64 Abort(fault);
        end;
    end;

    // All observers in the shareability domain observe the following load and store atomically.
    var memstatus : PhysMemRetStatus;
    (memstatus, oldvalue) = PhysMemRead{size}(memaddrdesc, accdesc);
    // Depending on the memory type of the physical address, the access might generate
    // either a synchronous External abort or an SError exception
    // among other CONSTRAINED UNPREDICTABLE choices.

    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
    end;

    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse{size}(oldvalue);
    end;

    var cmpfail : boolean = FALSE;
    var retvalue : bits(size) = oldvalue;
    if accdesc.acctype == AccessType FP then
        newvalue = MemAtomicFP{size}(accdesc.modop, oldvalue, operand);
    else
        (newvalue, cmpfail, retvalue) = MemAtomicInt{size}(accdesc.modop, oldvalue,
                                                         operand, cmpoperand);
    end;

    let requirerewrite : boolean = (!cmpfail ||
                                     ConstrainUnpredictableBool(Unpredictable WRITEFAILED CAS));

    if IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && StoreOnlyTagCheckingEnabled(accdesc.el) then
        // If the compare on a CAS fails, then it is CONSTRAINED UNPREDICTABLE whether the

```



```

// Tag check is performed.
if accdesc.tagchecked && !requirewrite then
    accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable STRONLYTAGCHECKEDCAS);
end;

if accdesc.tagchecked then
    let ltag : bits(4) = AArch64\_LogicalAddressTag(address);
    let readcheck : boolean = FALSE;
    var fault : FaultRecord = AArch64\_CheckTag(memaddrdesc, accdesc, readcheck,
                                             bytes, ltag);

    if fault.statuscode != Fault\_None then
        // For a synchronous Tag Check Fault due to FEAT_MTE_STORE_ONLY, set WnR.
        fault.write = TRUE;
        AArch64\_Abort(fault);
    end;
end;

if requirewrite then
    if BigEndian(accdesc.acctype) then
        newvalue = BigEndianReverse{size}(newvalue);
    end;
    memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, newvalue);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
    end;
end;

if SPESampleInFlight then
    let is_load : boolean = FALSE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);
end;

// Load operations return the old (pre-operation) value.
// Compare and Swap operations return the old (pre-operation) value. For a successful CAS,
// this might be the value from the compare operand or from memory.
return retvalue;
end;

```

Library pseudocode for aarch64/functions/memory/MemAtomicFP

```

// MemAtomicFP()
// =====
// Performs FP Atomic operation

func MemAtomicFP{N}(modop : MemAtomicOp, op1 : bits(N), op2 : bits(N)) => bits(N)
begin
    var fpcr : FPCR_Type = FPCR();
    let altfp : boolean = FALSE;
    let fpexc : boolean = FALSE;
    fpcr.[AH,DN] = '01';
    fpcr.FZ = fpcr.FZ OR fpcr.FIZ; // Treat FPCR.FIZ as equivalent to FPCR.FZ
    var result : bits(N);

    case modop of
        when MemAtomicOp\_FPADD => result = FPAdd{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_FPMAX => result = FPMax{N}(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp\_FPMIN => result = FPMin{N}(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp\_FPMAXNM => result = FPMaxNum{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_FPMINNM => result = FPMinNum{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_BFADD => result = BFAdd{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_BFMAX => result = BFMax{N}(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp\_BFMIN => result = BFMin{N}(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp\_BFMAXNM => result = BFMaxNum{N}(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_BFMINNM => result = BFMinNum{N}(op1, op2, fpcr, fpexc);
    end;
    return result;
end;

```

Library pseudocode for aarch64/functions/memory/MemAtomicInt

```
// MemAtomicInt()
// =====
// Performs Integer Atomic operation

func MemAtomicInt{N}(modop : MemAtomicOp, op1 : bits(N), op2 : bits(N), cmpop : bits(N)
    ) => (bits(N), boolean, bits(N))
begin
    var result : bits(N);
    var cmpfail : boolean = FALSE;
    var retvalue : bits(N) = op1;

    case modop of
        when MemAtomicOp\_ADD =>      result = op1 + op2;
        when MemAtomicOp\_BIC =>      result = op1 AND NOT(op2);
        when MemAtomicOp\_EOR =>      result = op1 XOR op2;
        when MemAtomicOp\_ORR =>      result = op1 OR op2;
        when MemAtomicOp\_SMAX =>     result = Max(SInt(op1), SInt(op2)) [N-1:0];
        when MemAtomicOp\_SMIN =>     result = Min(SInt(op1), SInt(op2)) [N-1:0];
        when MemAtomicOp\_UMAX =>     result = Max(UInt(op1), UInt(op2)) [N-1:0];
        when MemAtomicOp\_UMIN =>     result = Min(UInt(op1), UInt(op2)) [N-1:0];
        when MemAtomicOp\_SWP  =>      result = op2;
        when MemAtomicOp\_CAS  =>      (result, cmpfail, retvalue) = CASCompare{N}(op1, cmpop, op2);
        when MemAtomicOp\_GCSSS1 =>    (result, cmpfail, retvalue) = CASCompare{N}(op1, cmpop, op2);
    end;
    return (result, cmpfail, retvalue);
end;
```



```

// MemAtomicRCW()
// =====
// Perform a single-copy-atomic access with Read-Check-Write operation

func MemAtomicRCW{size : integer{64, 128}}(address : bits(64), cmpoperand : bits(size),
                                           operand : bits(size), accdesc_in : AccessDescriptor
                                           ) => (bits(4), bits(size))

begin
    assert accdesc_in.atomicop;
    assert accdesc_in.rcw;

    let bytes : integer{} = (size DIV 8) as integer{8, 16};

    var nzcw : bits(4);
    var oldvalue : bits(size);
    var newvalue : bits(size);
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64 AccessIsTagChecked(address, accdesc);
    end;

    if !aligned && AArch64 UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64 Abort(fault);
    end;

    // MMU or MPU lookup
    let memaddrdesc : AddressDescriptor = AArch64 TranslateAddress(address, accdesc,
                                                                aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64 Abort(memaddrdesc.fault);
    end;

    if (!IsWBShareable(memaddrdesc.memattrs) &&
        ConstrainUnpredictableBool(Unpredictable Atomic NOP)) then
        return (ARBITRARY : bits(4), ARBITRARY : bits(size));
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
    end;

    // For Store-only Tag checking, the tag check is performed on the store.
    if (accdesc.tagchecked &&
        (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
         !StoreOnlyTagCheckingEnabled(accdesc.el))) then
        let ltag : bits(4) = AArch64 LogicalAddressTag(address);
        let readcheck : boolean = TRUE;
        let fault : FaultRecord = AArch64 CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault None then
            AArch64 Abort(fault);
        end;
    end;

    // All observers in the shareability domain observe the following load and store atomically.
    var memstatus : PhysMemRetStatus;
    (memstatus, oldvalue) = PhysMemRead{size}(memaddrdesc, accdesc);
    // Depending on the memory type of the physical address, the access might generate
    // either a synchronous External abort or an SError exception
    // among other CONSTRAINED UNPREDICTABLE choices.

    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
    end;

    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse{size}(oldvalue);
    end;

    var cmpfail : boolean = FALSE;
    var retvalue : bits(size) = oldvalue;
    case accdesc.modop of
        when MemAtomicOp BIC => newvalue = oldvalue AND NOT(operand);
        when MemAtomicOp ORR => newvalue = oldvalue OR operand;
        when MemAtomicOp SWP => newvalue = operand;
        when MemAtomicOp CAS =>
            (newvalue, cmpfail, retvalue) = CASCompare{size}(oldvalue, cmpoperand, operand);
    end;

    if cmpfail then
        nzcw = '1010'; // N = 1 indicates compare failure
    else
        nzcw = RCWCheck{size}(retvalue, newvalue, accdesc.rcws);
    end;
end;

```

```

end;

// If RCWCheck() passes, it returns nzcw == '0010'
let requirewrite : boolean = (nzcw == '0010' ||
    ConstrainUnpredictableBool(Unpredictable\_WRITEFAILEDRCWCAS));

if IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && StoreOnlyTagCheckingEnabled(accdesc.el) then
    // If the RCW(S) check fails, or if the compare on a RCW(S) CAS fails,
    // then it is CONSTRAINED UNPREDICTABLE whether the Tag check is performed.
    if accdesc.tagchecked && !requirewrite then
        accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable\_STRONLYTAGCHECKEDRCWSCAS);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64\_LogicalAddressTag(address);
        let readcheck : boolean = FALSE;
        var fault : FaultRecord = AArch64\_CheckTag(memaddrdesc, accdesc, readcheck,
            bytes, ltag);

        if fault.statuscode != Fault\_None then
            // For a synchronous Tag Check Fault due to FEAT_MTE_STORE_ONLY, set WnR.
            fault.write = TRUE;
            AArch64\_Abort(fault);
        end;
    end;
end;

if requirewrite then
    if BigEndian(accdesc.acctype) then
        newvalue = BigEndianReverse{size}(newvalue);
    end;

    memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, newvalue);

    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
    end;
end;

if SPESampleInFlight then
    let is_load : boolean = TRUE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);
end;

// Load operations return the old (pre-operation) value.
// Compare and Swap operations return the old (pre-operation) value. For a successful CAS,
// this might be the value from the compare operand or from memory.
return (nzcw, retvalue);
end;

```



```

// MemLoad64B()
// =====
// Performs an atomic 64-byte read from a given virtual address.

func MemLoad64B(address : bits(64), accdesc_in : AccessDescriptor) => bits(512)
begin
    let size : integer{} = 512;
    let bytes : integer{} = size DIV 8;
    var data : bits(size);
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    if !aligned && AArch64 UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64 Abort(fault);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64 AccessIsTagChecked(address, accdesc);
    end;

    var memaddrdesc : AddressDescriptor = AArch64 TranslateAddress(address, accdesc,
                                                                aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64 Abort(memaddrdesc.fault);
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64 LogicalAddressTag(address);
        let readcheck : boolean = TRUE;
        let fault : FaultRecord = AArch64 CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault None then
            AArch64 Abort(fault);
        end;
    end;

    var byte_atomic : boolean = FALSE;
    if (memaddrdesc.memattrs.memtype == MemType Device ||
        (memaddrdesc.memattrs.inner.attrs == MemAttr NC &&
         memaddrdesc.memattrs.outer.attrs == MemAttr NC)) &&
        !AddressSupportsLS64(memaddrdesc.paddress.address) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable LS64UNSUPPORTED);
        assert c IN {Constraint LIMITED ATOMICITY, Constraint FAULT};
        if c == Constraint FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            let fault : FaultRecord = ExclusiveFault(accdesc, address);
            AArch64 Abort(fault);
        else
            byte_atomic = TRUE;
        end;
    elseif IsWBShareable(memaddrdesc.memattrs) && !IsConventionalMemory(memaddrdesc) then
        if ImpDefBool("LD64B faults to iWBtoWB non-Conventional memory") then
            // Generate a Data Abort reported using the DFSC code of 110101.
            let fault : FaultRecord = ExclusiveFault(accdesc, address);
            AArch64 Abort(fault);
        else
            byte_atomic = TRUE;
        end;
    end;

    if SPESampleInFlight then
        let is_load : boolean = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;

    var memstatus : PhysMemRetStatus;
    if byte_atomic then
        // Accesses are not single-copy atomic above the byte level.
        for i = 0 to bytes-1 do
            (memstatus, data[i*8]) = PhysMemRead{8}(memaddrdesc, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
            end;
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress = memaddrdesc.vaddress + 1;
        end;
    else
        (memstatus, data) = PhysMemRead{size}(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;
    end;
end;

```

```

        end;
    end;

    return data;
end;

```

Library pseudocode for aarch64/functions/memory/MemSingleGranule

```

// MemSingleGranule()
// =====
// When FEAT_LSE2 is implemented, for some memory accesses if all bytes
// of the accesses are within 16-byte quantity aligned to 16-bytes and
// satisfy additional requirements - then the access is guaranteed to
// be single copy atomic.
// However, when the accesses do not all lie within such a boundary, it
// is CONSTRAINED UNPREDICTABLE if the access is single copy atomic.
// In the pseudocode, this CONSTRAINED UNPREDICTABLE aspect is modeled via
// MemSingleGranule() which is IMPLEMENTATION DEFINED and, is at least 16 bytes
// and at most 4096 bytes.
// This is a limitation of the pseudocode.

func MemSingleGranule() => integer
begin
    let size : integer = ImpDefInt("Aligned quantity for atomic access");
    // access is assumed to be within 4096 byte aligned quantity to
    // avoid multiple translations for a single copy atomic access.
    assert (size >= 16) && (size <= 4096);
    return size;
end;

```



```

// MemStore64B()
// =====
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.

func MemStore64B(address : bits(64), value : bits(512), accdesc_in : AccessDescriptor)
begin
    let size : integer{} = 512;
    let bytes : integer{} = size DIV 8;
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    if !aligned && AArch64 UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64 Abort(fault);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64 AccessIsTagChecked(address, accdesc);
    end;

    var memaddrdesc : AddressDescriptor = AArch64 TranslateAddress(address, accdesc,
                                                                aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64 Abort(memaddrdesc.fault);
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64 LogicalAddressTag(address);
        let readcheck : boolean = FALSE;
        let fault : FaultRecord = AArch64 CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault None then
            AArch64 Abort(fault);
        end;
    end;

    var byte_atomic : boolean = FALSE;
    if (memaddrdesc.memattrs.memtype == MemType Device ||
        (memaddrdesc.memattrs.inner.attrs == MemAttr NC &&
         memaddrdesc.memattrs.outer.attrs == MemAttr NC)) &&
        !AddressSupportsLS64(memaddrdesc.paddress.address) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable LS64UNSUPPORTED);
        assert c IN {Constraint LIMITED ATOMICITY, Constraint FAULT};
        if c == Constraint FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            let fault : FaultRecord = ExclusiveFault(accdesc, address);
            AArch64 Abort(fault);
        else
            byte_atomic = TRUE;
        end;
    elseif IsWBShareable(memaddrdesc.memattrs) && !IsConventionalMemory(memaddrdesc) then
        if ImpDefBool("ST64B faults to iWBWB non-Conventional memory") then
            // Generate a Data Abort reported using the DFSC code of 110101.
            let fault : FaultRecord = ExclusiveFault(accdesc, address);
            AArch64 Abort(fault);
        else
            byte_atomic = TRUE;
        end;
    end;

    if SPESampleInFlight then
        let is_load : boolean = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;

    var memstatus : PhysMemRetStatus;
    if byte_atomic then
        // Accesses are not single-copy atomic above the byte level.
        for i = 0 to bytes-1 do
            memstatus = PhysMemWrite{8}(memaddrdesc, accdesc, value[i*8]);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            end;
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress = memaddrdesc.vaddress + 1;
        end;
    else
        memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
        end;
    end;
end;

```

```

    end;
end;

return;
end;

```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRet

```

// MemStore64BWithRet()
// =====
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

func MemStore64BWithRet(address : bits(64), value : bits(512),
    accdesc_in : AccessDescriptor) => bits(64)
begin
    let size : integer{} = 512;
    let bytes : integer{} = size DIV 8;
    var accdesc : AccessDescriptor = accdesc_in;
    let aligned : boolean = IsAlignedSize(address, bytes);

    if !aligned && AArch64\_UnalignedAccessFaults(accdesc, address, bytes) then
        let fault : FaultRecord = AlignmentFault(accdesc, address);
        AArch64\_Abort(fault);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64\_AccessIsTagChecked(address, accdesc);
    end;

    let memaddrdesc : AddressDescriptor = AArch64\_TranslateAddress(address, accdesc,
        aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64\_Abort(memaddrdesc.fault);
    end;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64\_LogicalAddressTag(address);
        let readcheck : boolean = FALSE;
        let fault : FaultRecord = AArch64\_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault\_None then
            AArch64\_Abort(fault);
        end;
    end;

    var memstatus : PhysMemRetStatus;
    memstatus = PhysMemWrite{size}(memaddrdesc, accdesc, value);

    // If an access generated by ST64BV or ST64BV0 fails solely because the memory location
    // does not support 64-byte access, then memstatus does not indicate a fault,
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
    end;

    if SPESampleInFlight then
        let is_load : boolean = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);
    end;

    return memstatus.store64bstatus;
end;

```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRetStatus

```

// MemStore64BWithRetStatus()
// =====
// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.

impdef func MemStore64BWithRetStatus() => bits(64)
begin
    return Zeros{64};
end;

```

Library pseudocode for aarch64/functions/memory/NVMem

```
// NVMem - accessor
// =====

accessor NVMem(offset : integer) <=> value : bits(64)
begin
    // This function is the load memory access for the transformed System register read access
    // when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
    // The address for the load memory access is calculated using
    // the formula SignExtend{64}(VNCR_EL2.BADDR :: Offset[11:0]) where,
    // * VNCR_EL2.BADDR holds the base address of the memory location, and
    // * Offset is the unique offset value defined architecturally for each System register that
    // supports transformation of register access to memory access.
    getter
        assert offset > 0 && offset MOD 8 == 0;
        let directread : boolean = FALSE;
        let baddr : bits(64) = EffectiveBADDR(VNCR_EL2().BADDR :: Zeros{12}, directread);
        let address : bits(64) = baddr + offset;
        let accdesc : AccessDescriptor = CreateAccDescNV2(MemOp_LOAD);
        return Mem{64}(address, accdesc);
    end;

    // This function is the store memory access for the transformed System register write access
    // when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
    // The address for the store memory access is calculated using
    // the formula SignExtend{64}(VNCR_EL2.BADDR :: Offset[11:0]) where,
    // * VNCR_EL2.BADDR holds the base address of the memory location, and
    // * Offset is the unique offset value defined architecturally for each System register that
    // supports transformation of register access to memory access.
    setter
        assert offset > 0 && offset MOD 8 == 0;
        let directread : boolean = FALSE;
        let baddr : bits(64) = EffectiveBADDR(VNCR_EL2().BADDR :: Zeros{12}, directread);
        let address : bits(64) = baddr + offset;
        let accdesc : AccessDescriptor = CreateAccDescNV2(MemOp_STORE);
        Mem{64}(address, accdesc) = value;
        return;
    end;
end;

accessor NVMem128(offset : integer) <=> value : bits(128)
begin
    getter
        assert offset > 0 && offset MOD 16 == 0;
        let directread : boolean = FALSE;
        let baddr : bits(64) = EffectiveBADDR(VNCR_EL2().BADDR :: Zeros{12}, directread);
        let address : bits(64) = baddr + offset;
        let accdesc : AccessDescriptor = CreateAccDescNV2(MemOp_LOAD);
        return Mem{128}(address, accdesc);
    end;

    setter
        assert offset > 0 && offset MOD 16 == 0;
        let directread : boolean = FALSE;
        let baddr : bits(64) = EffectiveBADDR(VNCR_EL2().BADDR :: Zeros{12}, directread);
        let address : bits(64) = baddr + offset;
        let accdesc : AccessDescriptor = CreateAccDescNV2(MemOp_STORE);
        Mem{128}(address, accdesc) = value;
    end;
end;
```

Library pseudocode for aarch64/functions/memory/PhysMemTagRead

```
// PhysMemTagRead()
// =====
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

impdef func PhysMemTagRead(desc : AddressDescriptor,
                           accdesc : AccessDescriptor) => (PhysMemRetStatus, bits(4))
begin
    return (ARBITRARY : PhysMemRetStatus, Zeros{4});
end;
```

Library pseudocode for aarch64/functions/memory/PhysMemTagWrite

```
// PhysMemTagWrite()
// =====
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

impdef func PhysMemTagWrite(desc : AddressDescriptor,
                           accdesc : AccessDescriptor,
                           value : bits(4)) => PhysMemRetStatus

begin
    return ARBITRARY : PhysMemRetStatus;
end;
```

Library pseudocode for aarch64/functions/memory/StoreOnlyTagCheckingEnabled

```
// StoreOnlyTagCheckingEnabled()
// =====
// Returns TRUE if loads executed at the given Exception level are Tag unchecked.

func StoreOnlyTagCheckingEnabled(el : bits(2)) => boolean
begin
    assert IsFeatureImplemented(FEAT_MTE_STORE_ONLY);
    var tcso : bit;

    case el of
        when EL0 =>
            if !ELIsInHost(el) then
                tcso = SCTL_EL1().TCSO0;
            else
                tcso = SCTL_EL2().TCSO0;
            end;
        when EL1 =>
            tcso = SCTL_EL1().TCSO;
        when EL2 =>
            tcso = SCTL_EL2().TCSO;
        otherwise =>
            tcso = SCTL_EL3().TCSO;
    end;

    return tcso == '1';
end;
```

Library pseudocode for aarch64/functions/mops/ArchMaxMOPSBlockSize

```
// ArchMaxMOPSBlockSize
// =====
// Maximum number of bytes CPY/SET instructions can use

constant ArchMaxMOPSBlockSize : integer{} = 0x7FFF_FFFF_FFFF_FFFF;
```

Library pseudocode for aarch64/functions/mops/ArchMaxMOPSCPysize

```
// ArchMaxMOPSCPysize
// =====
// Maximum number of bytes CPY instructions can use

constant ArchMaxMOPSCPysize : integer{} = 0x007F_FFFF_FFFF_FFFF;
```

Library pseudocode for aarch64/functions/mops/ArchMaxMOPSETGSize

```
// ArchMaxMOPSETGSize
// =====
// Maximum number of bytes SETG instructions can use

constant ArchMaxMOPSETGSize : integer{} = 0x7FFF_FFFF_FFFF_FFF0;
```

Library pseudocode for aarch64/functions/mops/CPYFOptionA

```
// CPYFOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPYF* instructions, and FALSE otherwise.

func CPYFOptionA() => boolean
begin
    return ImpDefBool("CPYF* instructions use Option A");
end;
```

Library pseudocode for aarch64/functions/mops/CPYOptionA

```
// CPYOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPY* instructions, and FALSE otherwise.

func CPYOptionA() => boolean
begin
    return ImpDefBool("CPY* instructions use Option A");
end;
```

Library pseudocode for aarch64/functions/mops/CPYParams

```
// CPYParams
// =====

type CPYParams of record {
    stage : MOPSSStage,
    implements_option_a : boolean,
    forward : boolean,
    cpysize : integer,
    stagecpysize : integer,
    toaddress : bits(64),
    fromaddress : bits(64),
    nzcvc : bits(4),
    n : integer,
    d : integer,
    s : integer
};
```

Library pseudocode for aarch64/functions/mops/CPYPostSizeChoice

```
// CPYPostSizeChoice()
// =====
// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.

impdef func CPYPostSizeChoice(memcpy : CPYParams) => integer
begin
    return ARBITRARY : integer;
end;
```

Library pseudocode for aarch64/functions/mops/CPYPreSizeChoice

```
// CPYPreSizeChoice()
// =====
// Returns the size of the copy that is performed by the CPYP* instructions for this
// implementation given the parameters of the destination, source and size of the copy.

impdef func CPYPreSizeChoice(memcpy : CPYParams) => integer
begin
    return ARBITRARY : integer;
end;
```

Library pseudocode for aarch64/functions/mops/CPYSizeChoice

```
// CPYSizeChoice()
// =====
// Returns the size of the block this performed for an iteration of the copy given the
// parameters of the destination, source and size of the copy.

impdef func CPYSizeChoice(memcpy : CPYParams) => MOPSSBlockSize
begin
    return ARBITRARY : MOPSSBlockSize;
end;
```

Library pseudocode for aarch64/functions/mops/CheckCPYConstrainedUnpredictable

```
// CheckCPYConstrainedUnpredictable()
// =====
// Check for CONSTRAINED UNPREDICTABLE behavior in the CPY* and CPYF* instructions.

func CheckCPYConstrainedUnpredictable(n : integer, d : integer, s : integer)
begin
  if (s == n || s == d || n == d) then
    let c : Constraint = ConstrainUnpredictable(Unpredictable MOPSOVERLAP);
    assert c IN {Constraint UNDEF, Constraint NOP};
    case c of
      when Constraint UNDEF =>      Undefined();
      when Constraint NOP  =>      ExecuteAsNOP();
    end;
  end;

  if (d == 31 || s == 31 || n == 31) then
    let c : Constraint = ConstrainUnpredictable(Unpredictable MOPS R31);
    assert c IN {Constraint UNDEF, Constraint NOP};
    case c of
      when Constraint UNDEF =>      Undefined();
      when Constraint NOP  =>      ExecuteAsNOP();
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/mops/CheckMOPSEnabled

```
// CheckMOPSEnabled()
// =====
// Check for EL0 and EL1 access to the CPY* and SET* instructions.

func CheckMOPSEnabled()
begin
  if (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELIsInHost(EL0) &&
    (!IsHCRXEL2Enabled() || HCRX\_EL2().MSCEn == '0')) then
    Undefined();
  end;

  if PSTATE.EL == EL0 && !IsInHost() && SCTLR\_EL1().MSCEn == '0' then
    Undefined();
  end;

  if PSTATE.EL == EL0 && IsInHost() && SCTLR\_EL2().MSCEn == '0' then
    Undefined();
  end;
end;
```

Library pseudocode for aarch64/functions/mops/CheckMemCpyParams

```
// CheckMemCpyParams()
// =====
// Check if the parameters to a CPY* or CPYF* instruction are consistent with the
// PE state and well-formed.

func CheckMemCpyParams(memcpy : CPYParams, options : bits(4))
begin
  let from_epilogue : boolean = memcpy.stage == MOPSSStage Epilogue;

  // Check if this version is consistent with the state of the call.
  if ((memcpy.stagecpysize != 0 || MemStageCpyZeroSizeCheck()) &&
    (memcpy.cpysize != 0 || MemCpyZeroSizeCheck())) then
    let using_option_a : boolean = memcpy.nzcv[1] == '0';
    if memcpy.implements_option_a != using_option_a then
      let formatoption : bits(2) = '1':: (if memcpy.implements_option_a then '1' else '0');
      MismatchedMemCpyException(memcpy, options, formatoption);
    end;
  end;

  // Check if the parameters to this instruction are valid.
  if memcpy.stage == MOPSSStage Main then
    if MemCpyParametersIllformedM(memcpy) then
      let formatoption : bits(2) = '0':: (if memcpy.implements_option_a then '1' else '0');
      MismatchedMemCpyException(memcpy, options, formatoption);
    end;
  else
    let postsize : integer = CPYPostSizeChoice(memcpy);
    if memcpy.cpysize != postsize || MemCpyParametersIllformedE(memcpy) then
      let formatoption : bits(2) = '0':: (if memcpy.implements_option_a then '1' else '0');
      MismatchedMemCpyException(memcpy, options, formatoption);
    end;
  end;

  return;
end;
```

Library pseudocode for aarch64/functions/mops/CheckMemSetParams

```
// CheckMemSetParams ()
// =====
// Check if the parameters to a SET* or SETG* instruction are consistent with the
// PE state and well-formed.

func CheckMemSetParams(memset : SETParams, options : bits(2))
begin
    let from_epilogue : boolean = memset.stage == MOPSSStage Epilogue;

    // Check if this version is consistent with the state of the call.
    if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
        (memset.setsize != 0 || MemSetZeroSizeCheck())) then
        let using_option_a : boolean = memset.nzcv[1] == '0';
        if memset.implements_option_a != using_option_a then
            let formatoption : bits(2) = '1': (if memset.implements_option_a then '1' else '0');
            MismatchedMemSetException(memset, options, formatoption);
        end;
    end;

    // Check if the parameters to this instruction are valid.
    if memset.stage == MOPSSStage Main then
        if MemSetParametersIllformedM(memset) then
            let formatoption : bits(2) = '0': (if memset.implements_option_a then '1' else '0');
            MismatchedMemSetException(memset, options, formatoption);
        end;
    else
        let postsize : integer = SETPostSizeChoice(memset);
        if memset.setsize != postsize || MemSetParametersIllformedE(memset) then
            let formatoption : bits(2) = '0': (if memset.implements_option_a then '1' else '0');
            MismatchedMemSetException(memset, options, formatoption);
        end;
    end;

    return;
end;
```

Library pseudocode for aarch64/functions/mops/CheckSETConstrainedUnpredictable

```
// CheckSETConstrainedUnpredictable()
// =====
// Check for CONSTRAINED UNPREDICTABLE behavior in the SET* and SETG* instructions.

func CheckSETConstrainedUnpredictable(memset : SETParams)
begin
    if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable MOPSOVERLAP);
        assert c IN {Constraint UNDEF, Constraint NOP};
        case c of
            when Constraint UNDEF => Undefined();
            when Constraint NOP   => ExecuteAsNOP();
        end;
    end;

    if (memset.d == 31 || memset.n == 31) then
        let c : Constraint = ConstrainUnpredictable(Unpredictable MOPS R31);
        assert c IN {Constraint UNDEF, Constraint NOP};
        case c of
            when Constraint UNDEF => Undefined();
            when Constraint NOP   => ExecuteAsNOP();
        end;
    end;

end;
```


Library pseudocode for aarch64/functions/mops/IsMemCpyForward

```
// IsMemCpyForward()
// =====
// Returns TRUE if in a memcpy of size cpysize bytes from the source address fromaddress
// to destination address toaddress is done in the forward direction on this implementation.

func IsMemCpyForward(memcpy : CPYParams) => boolean
begin
    var forward : boolean;

    // Check for overlapping cases
    if ((UInt(memcpy.fromaddress[55:0]) > UInt(memcpy.toaddress[55:0])) &&
        (UInt(memcpy.fromaddress[55:0]) < UInt(ZeroExtend{64}(memcpy.toaddress[55:0]) +
            memcpy.cpysize))) then
        forward = TRUE;

    elsif ((UInt(memcpy.fromaddress[55:0]) < UInt(memcpy.toaddress[55:0])) &&
        (UInt(ZeroExtend{64}(memcpy.fromaddress[55:0]) + memcpy.cpysize) >
            UInt(memcpy.toaddress[55:0]))) then
        forward = FALSE;

    // Non-overlapping case
    else
        forward = ImpDefBool("CPY in the forward direction");
    end;

    return forward;
end;
```

Library pseudocode for aarch64/functions/mops/MOPSBlockSize

```
// MOPSBlockSize
// =====

type MOPSBlockSize of integer{0..MaxMOPSBlockSize};
```

Library pseudocode for aarch64/functions/mops/MOPSStage

```
// MOPSStage
// =====

type MOPSStage of enumeration { MOPSStage_Prologue, MOPSStage_Main, MOPSStage_Epilogue };
```

Library pseudocode for aarch64/functions/mops/MaxBlockSizeCopiedBytes

```
// MaxBlockSizeCopiedBytes()
// =====
// Returns the maximum number of bytes that can used in a single block of the copy.

func MaxBlockSizeCopiedBytes() => integer
begin
    return ImpDefInt("Maximum bytes used in a single block of a copy");
end;
```

Library pseudocode for aarch64/functions/mops/MaxMOPSBlockSize

```
// MaxMOPSBlockSize
// =====
// Maximum number of bytes CPY/SET instructions can use

config MaxMOPSBlockSize : integer{1..ArchMaxMOPSBlockSize} = 64;
```



```

// MemCpyBytes()
// =====
// Copies 'bytes' bytes of memory from fromaddress to toaddress.
// The integer return parameter indicates the number of bytes copied. The boolean return parameter
// indicates if a Fault or Abort occurred on the write. The AddressDescriptor and PhysMemRetStatus
// parameters contain Fault or Abort information for the caller to handle.

func MemCpyBytes(toaddress : bits(64),
                fromaddress : bits(64),
                forward : boolean,
                bytes : MOPSBBlockSize,
                raccdesc : AccessDescriptor,
                waccdesc : AccessDescriptor
                ) => (integer, boolean, AddressDescriptor, PhysMemRetStatus)

begin
    var rmemaddrdesc : AddressDescriptor;           // AddressDescriptor for reads
    var rmemstatus : PhysMemRetStatus;             // PhysMemRetStatus for writes
    rmemaddrdesc.fault = NoFault();
    rmemstatus.statuscode = Fault_None;

    var wmemaddrdesc : AddressDescriptor;           // AddressDescriptor for writes
    var wmemstatus : PhysMemRetStatus;             // PhysMemRetStatus for writes
    wmemaddrdesc.fault = NoFault();
    wmemstatus.statuscode = Fault_None;

    var value : bits(8*bytes);
    let aligned : boolean = TRUE;

    if forward then
        var read : integer = 0;                    // Bytes read
        var write : integer = 0;                   // Bytes written

        // Read until all bytes are read or until a fault is encountered.
        while (read < bytes && !IsFault(rmemaddrdesc) && !IsFault(rmemstatus))
            looplimit MaxMOPSBBlockSize do
                (value[8 * read +:8], rmemaddrdesc, rmemstatus) = AArch64_MemSingleRead{8}(
                                                                    fromaddress + read,
                                                                    raccdesc, aligned);

                read = read + 1;
            end;

        // Ensure no UNKNOWN data is written.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            read = read - 1;
        end;

        // Write all bytes that were read or until a fault is encountered.
        while write < read && !IsFault(wmemaddrdesc) && !IsFault(wmemstatus)
            looplimit MaxMOPSBBlockSize do
                (wmemaddrdesc, wmemstatus) = AArch64_MemSingleWrite{8}(toaddress + write,
                                                                    waccdesc, aligned,
                                                                    value[8 * write +:8]);

                write = write + 1;
            end;

        // Check all bytes were written.
        if IsFault(wmemaddrdesc) || IsFault(wmemstatus) then
            let fault_on_write : boolean = TRUE;
            return (write - 1, fault_on_write, wmemaddrdesc, wmemstatus);
        end;

        // Check all bytes were read.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            let fault_on_write : boolean = FALSE;
            return (read, fault_on_write, rmemaddrdesc, rmemstatus);
        end;

    else
        var read : integer = bytes;                // Bytes to read
        var write : integer = bytes;               // Bytes to write

        // Read until all bytes are read or until a fault is encountered.
        while read > 0 && !IsFault(rmemaddrdesc) && !IsFault(rmemstatus)
            looplimit MaxMOPSBBlockSize do
                read = read - 1;
                (value[8 * read +:8], rmemaddrdesc, rmemstatus) = AArch64_MemSingleRead{8}(
                                                                    fromaddress + read,
                                                                    raccdesc, aligned);
            end;

        // Ensure no UNKNOWN data is written.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            read = read + 1;
        end;

        // Write all bytes that were read or until a fault is encountered.
        while write > read && !IsFault(wmemaddrdesc) && !IsFault(wmemstatus)
            looplimit MaxMOPSBBlockSize do
                write = write - 1;
            end;
    end
end

```

```

        (wmemaddrdesc, wmemstatus) = AArch64\_MemSingleWrite{8}(toaddress + write,
                                                                    waccdesc, aligned,
                                                                    value[8 * write +:8]);
    end;

    // Check all bytes were written.
    if IsFault(wmemaddrdesc) || IsFault(wmemstatus) then
        let fault_on_write : boolean = TRUE;
        return (bytes - (write + 1), fault_on_write, wmemaddrdesc, wmemstatus);
    end;

    // Check all bytes were read.
    if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
        let fault_on_write : boolean = FALSE;
        return (bytes - read, fault_on_write, rmemaddrdesc, rmemstatus);
    end;
end;

// Return any AddressDescriptor and PhysMemRetStatus.
return (bytes, FALSE, wmemaddrdesc, wmemstatus);
end;

```

Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedE

```

// MemCpyParametersIllformedE()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.

impdef func MemCpyParametersIllformedE(memcpy : CPYParams) => boolean
begin
    return FALSE;
end;

```

Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedM

```

// MemCpyParametersIllformedM()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPM* instruction for this implementation given the parameters of the destination,
// source and size of the copy.

impdef func MemCpyParametersIllformedM(memcpy : CPYParams) => boolean
begin
    return FALSE;
end;

```

Library pseudocode for aarch64/functions/mops/MemCpyStageSize

```

// MemCpyStageSize()
// =====
// Returns the number of bytes copied by the given stage of a CPY* or CPM* instruction.

func MemCpyStageSize(memcpy : CPYParams) => integer
begin
    var stagecpysize : integer;

    if memcpy.stage == MOPSTage\_Prologue then
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(memcpy);
        assert stagecpysize == 0 || (stagecpysize < 0) == (memcpy.cpsize < 0);

        if memcpy.cpsize > 0 then
            assert stagecpysize <= memcpy.cpsize;
        else
            assert stagecpysize >= memcpy.cpsize;
        end;
    else
        let postsize : integer = CPYPostSizeChoice(memcpy);
        assert postsize == 0 || (postsize < 0) == (memcpy.cpsize < 0);

        if memcpy.stage == MOPSTage\_Main then
            stagecpysize = memcpy.cpsize - postsize;
        else
            stagecpysize = postsize;
        end;
    end;

    return stagecpysize;
end;

```

Library pseudocode for aarch64/functions/mops/MemCpyZeroSizeCheck

```
// MemCpyZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.

func MemCpyZeroSizeCheck() => boolean
begin
    return ImpDefBool("Implementation option is checked with a cpy size of 0");
end;
```

Library pseudocode for aarch64/functions/mops/MemSetBytes

```
// MemSetBytes()
// =====
// Writes a byte of data to the given address 'bytes' times.
// The integer return parameter indicates the number of bytes set. The AddressDescriptor and
// PhysMemRetStatus parameters contain Fault or Abort information for the caller to handle.

func MemSetBytes(toaddress : bits(64), data : bits(8),
    bytes : MOPSBlockSize,
    accdesc : AccessDescriptor) => (integer, AddressDescriptor, PhysMemRetStatus)
begin
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    memaddrdesc.fault = NoFault();
    memstatus.statuscode = Fault None;

    let aligned : boolean = TRUE;
    var write : integer = 0;                                // Bytes written

    // Write until all bytes are written or a fault is encountered.
    while write < bytes && !IsFault(memaddrdesc) && !IsFault(memstatus)
        looplimit MaxMOPSBlockSize do
            (memaddrdesc, memstatus) = AArch64 MemSingleWrite{8}(toaddress + write, accdesc,
                aligned, data);
            write = write + 1;
        end;

    // Check all bytes were written.
    if IsFault(memaddrdesc) || IsFault(memstatus) then
        return (write - 1, memaddrdesc, memstatus);
    end;

    return (bytes, memaddrdesc, memstatus);
end;
```

Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedE

```
// MemSetParametersIllformedE()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.

impdef func MemSetParametersIllformedE(memset : SETParams) => boolean
begin
    return FALSE;
end;
```

Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedM

```
// MemSetParametersIllformedM()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.

impdef func MemSetParametersIllformedM(memset : SETParams) => boolean
begin
    return FALSE;
end;
```

Library pseudocode for aarch64/functions/mops/MemSetStageSize

```
// MemSetStageSize()
// =====
// Returns the number of bytes set by the given stage of a SET* or SETG* instruction.

func MemSetStageSize(memset : SETParams) => integer
begin
    var stagesetsize : integer;

    if memset.stage == MOPSTage\_Prologue then
        // IMP DEF selection of the amount covered by pre-processing.
        stagesetsize = SETPreSizeChoice(memset);
        assert stagesetsize == 0 || (stagesetsize < 0) == (memset.setsize < 0);

        if memset.is_setg then assert stagesetsize[3:0] == '0000'; end;

        if memset.setsize > 0 then
            assert stagesetsize <= memset.setsize;
        else
            assert stagesetsize >= memset.setsize;
        end;

    else
        let postsize : integer = SETPostSizeChoice(memset);
        assert postsize == 0 || (postsize < 0) == (memset.setsize < 0);
        if memset.is_setg then assert postsize[3:0] == '0000'; end;

        if memset.stage == MOPSTage\_Main then
            stagesetsize = memset.setsize - postsize;
        else
            stagesetsize = postsize;
        end;
    end;

    return stagesetsize;
end;
```

Library pseudocode for aarch64/functions/mops/MemSetTags

```
// MemSetTags()
// =====
// Write Allocation Tags for each Tag Granule in 'size'.
// The integer return parameter indicates the number of Tag Granules written. The
// AddressDescriptor and PhysMemRetStatus parameters contain Fault or Abort information for
// the caller to handle.

func MemSetTags(toaddress : bits(64), tag : bits(4),
    size : integer, accdesc : AccessDescriptor
) => (integer, AddressDescriptor, PhysMemRetStatus)
begin
    assert IsAlignedSize{64}(toaddress, TAG GRANULE) && size MOD TAG GRANULE == 0;

    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    memaddrdesc.fault = NoFault();
    memstatus.statuscode = Fault None;
    var tagstep : integer = size DIV TAG GRANULE;

    // Write until all tags are written or a fault is encountered.
    while tagstep > 0 && !IsFault(memaddrdesc) && !IsFault(memstatus)
        looplimit ArchMaxMOPSSETGSize DIV TAG GRANULE do
            let tagaddr : bits(64) = toaddress + (tagstep - 1) * TAG GRANULE;
            (memaddrdesc, memstatus) = AArch64\_MemTagWrite(tagaddr, accdesc, tag);
            tagstep = tagstep - 1;
        end;

    if IsFault(memaddrdesc) || IsFault(memstatus) then
        tagstep = tagstep + 1;
    end;

    let tags_written : integer = (size DIV TAG GRANULE) - tagstep;
    return (tags_written, memaddrdesc, memstatus);
end;
```

Library pseudocode for aarch64/functions/mops/MemSetZeroSizeCheck

```
// MemSetZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a set of size zero remaining.

func MemSetZeroSizeCheck() => boolean
begin
    return ImpDefBool("Implementation option is checked with a setsize of 0");
end;
```

Library pseudocode for aarch64/functions/mops/MemStageCpyZeroSizeCheck

```
// MemStageCpyZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a stage copy of size zero remaining.

func MemStageCpyZeroSizeCheck() => boolean
begin
    return (ImpDefBool(
        "Implementation option is checked with a stage cpysize of 0"));
end;
```

Library pseudocode for aarch64/functions/mops/MemStageSetZeroSizeCheck

```
// MemStageSetZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a stage set of size zero remaining.

func MemStageSetZeroSizeCheck() => boolean
begin
    return (ImpDefBool(
        "Implementation option is checked with a stage setsize of 0"));
end;
```

Library pseudocode for aarch64/functions/mops/MismatchedCpySetTargetEL

```
// MismatchedCpySetTargetEL()
// =====
// Return the target exception level for an Exception_MemCpyMemSet.

func MismatchedCpySetTargetEL() => bits(2)
begin
    var target_el : bits(2);

    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1' then
        target_el = EL2;
    elsif (PSTATE.EL == EL1 && EL2Enabled() &&
        IsHCRXEL2Enabled() && HCRX_EL2().MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1;
    end;

    return target_el;
end;
```

Library pseudocode for aarch64/functions/mops/MismatchedMemCpyException

```
// MismatchedMemCpyException()
// =====
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

func MismatchedMemCpyException(memcpy : CPYParams, options : bits(4), formatoption : bits(2))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
    let target_el : bits(2) = MismatchedCpySetTargetEL();

    var except : ExceptionRecord = ExceptionSyndrome(Exception_MemCpyMemSet);
    except.syndrome.iss[24] = '0';
    except.syndrome.iss[23] = '0';
    except.syndrome.iss[22:19] = options;
    except.syndrome.iss[18] = if memcpy.stage == MOPSStage_Epilogue then '1' else '0';
    except.syndrome.iss[17:16] = formatoption;
    // exception.syndrome[15] is RES0.
    except.syndrome.iss[14:10] = memcpy.d[4:0];
    except.syndrome.iss[9:5] = memcpy.s[4:0];
    except.syndrome.iss[4:0] = memcpy.n[4:0];

    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/functions/mops/MismatchedMemSetException

```
// MismatchedMemSetException()
// =====
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call.

func MismatchedMemSetException(memset : SETParams, options : bits(2), formatoption : bits(2))
begin
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;
    let target_el : bits(2) = MismatchedCpySetTargetEL();

    var except : ExceptionRecord = ExceptionSyndrome(Exception\_MemCpyMemSet);
    except.syndrome.iss[24] = '1';
    except.syndrome.iss[23] = if memset.is_setg then '1' else '0';
    // exception.syndrome[22:21] is RES0.
    except.syndrome.iss[20:19] = options;
    except.syndrome.iss[18] = if memset.stage == MOPSSStage Epilogue then '1' else '0';
    except.syndrome.iss[17:16] = formatoption;
    // exception.syndrome[15] is RES0.
    except.syndrome.iss[14:10] = memset.d[4:0];
    except.syndrome.iss[9:5] = memset.s[4:0];
    except.syndrome.iss[4:0] = memset.n[4:0];

    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/functions/mops/SETGOptionA

```
// SETGOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// SETG* instructions, and FALSE otherwise.

func SETGOptionA() => boolean
begin
    return ImpDefBool("SETG* instructions use Option A");
end;
```

Library pseudocode for aarch64/functions/mops/SETOptionA

```
// SETOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// SET* instructions, and FALSE otherwise.

func SETOptionA() => boolean
begin
    return ImpDefBool("SET* instructions use Option A");
end;
```

Library pseudocode for aarch64/functions/mops/SETParams

```
// SETParams
// =====

type SETParams of record {
    stage : MOPSSStage,
    implements_option_a : boolean,
    is_setg : boolean,
    setsize : integer,
    stagesetsize : integer,
    toaddress : bits(64),
    nzcv : bits(4),
    n : integer,
    d : integer,
    s : integer
};
```

Library pseudocode for aarch64/functions/mops/SETPostSizeChoice

```
// SETPostSizeChoice()
// =====
// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.

impdef func SETPostSizeChoice(memset : SETParams) => integer
begin
    return ARBITRARY : integer;
end;
```


Library pseudocode for aarch64/functions/mops/SETPreSizeChoice

```
// SETPreSizeChoice()
// =====
// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// for this implementation, given the parameters of the destination and size of the set.

impdef func SETPreSizeChoice(memset : SETParams) => integer
begin
    return ARBITRARY : integer;
end;
```

Library pseudocode for aarch64/functions/mops/SETSizeChoice

```
// SETSizeChoice()
// =====
// Returns the size of the block this performed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of alignsize.

impdef func SETSizeChoice(memset : SETParams, alignsize : integer) => MOPSBlockSize
begin
    return ARBITRARY : MOPSBlockSize;
end;
```

Library pseudocode for aarch64/functions/mops/UpdateCpyRegisters

```
// UpdateCpyRegisters()
// =====
// Performs updates to the X[n], X[d], and X[s] registers, as appropriate, for the CPY* and CPYF*
// instructions. When fault is TRUE, the values correspond to the first element not copied,
// such that a return to the instruction will enable a resumption of the copy.

func UpdateCpyRegisters(memcpy : CPYParams, fault : boolean, copied : integer)
begin
    if fault then
        if memcpy.stage == MOPSSStage Prologue then
            // Undo any formatting of the input parameters performed in the prologue.
            if memcpy.implements_option_a then
                if memcpy.forward then
                    // cpy size is negative.
                    let cpy size : integer = memcpy.cpy size + copied;
                    X{64}(memcpy.n) = (0 - cpy size)[63:0];
                    X{64}(memcpy.d) = memcpy.toaddress + cpy size;
                    X{64}(memcpy.s) = memcpy.fromaddress + cpy size;

                else
                    X{64}(memcpy.n) = (memcpy.cpy size - copied)[63:0];
                end;

            else
                if memcpy.forward then
                    X{64}(memcpy.n) = (memcpy.cpy size - copied)[63:0];
                    X{64}(memcpy.d) = memcpy.toaddress + copied;
                    X{64}(memcpy.s) = memcpy.fromaddress + copied;

                else
                    X{64}(memcpy.n) = (memcpy.cpy size - copied)[63:0];
                end;
            end;
        else
            if memcpy.implements_option_a then
                if memcpy.forward then
                    X{64}(memcpy.n) = (memcpy.cpy size + copied)[63:0];
                else
                    X{64}(memcpy.n) = (memcpy.cpy size - copied)[63:0];
                end;

            else
                X{64}(memcpy.n) = (memcpy.cpy size - copied)[63:0];

                if memcpy.forward then
                    X{64}(memcpy.d) = memcpy.toaddress + copied;
                    X{64}(memcpy.s) = memcpy.fromaddress + copied;
                else
                    X{64}(memcpy.d) = memcpy.toaddress - copied;
                    X{64}(memcpy.s) = memcpy.fromaddress - copied;
                end;
            end;
        end;
    else
        X{64}(memcpy.n) = memcpy.cpy size[63:0];
        if memcpy.stage == MOPSSStage Prologue || !memcpy.implements_option_a then
            X{64}(memcpy.d) = memcpy.toaddress;
            X{64}(memcpy.s) = memcpy.fromaddress;
        end;
    end;

    return;
end;
```

Library pseudocode for aarch64/functions/mops/UpdateSetRegisters

```
// UpdateSetRegisters()
// =====
// Performs updates to the X[n] and X[d] registers, as appropriate, for the SET* and SETG*
// instructions. When fault is TRUE, the values correspond to the first element not set, such
// that a return to the instruction will enable a resumption of the memory set.

func UpdateSetRegisters(memset : SETParams, fault : boolean, memory_set : integer)
begin
    if fault then
        // Undo any formatting of the input parameters performed in the prologue.
        if memset.stage == MOPSTage Prologue then
            if memset.implements_option_a then
                // setsize is negative.
                let setsize : integer = memset.setsize + memory_set;
                X{64}(memset.n) = (0 - setsize)[63:0];
                X{64}(memset.d) = memset.toaddress + setsize;
            else
                X{64}(memset.n) = (memset.setsize - memory_set)[63:0];
                X{64}(memset.d) = memset.toaddress + memory_set;
            end;
        else
            if memset.implements_option_a then
                X{64}(memset.n) = (memset.setsize + memory_set)[63:0];
            else
                X{64}(memset.n) = (memset.setsize - memory_set)[63:0];
                X{64}(memset.d) = memset.toaddress + memory_set;
            end;
        end;
    else
        X{64}(memset.n) = memset.setsize[63:0];
        if memset.stage == MOPSTage Prologue || !memset.implements_option_a then
            X{64}(memset.d) = memset.toaddress;
        end;
    end;

    return;
end;
```

Library pseudocode for aarch64/functions/movewideop/MoveWideOp

```
// MoveWideOp
// =====
// Move wide 16-bit immediate instruction types.

type MoveWideOp of enumeration {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

Library pseudocode for aarch64/functions/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// =====
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

func MoveWidePreferred(sf : bit, immN : bit, imms : bits(6), immr : bits(6)) => boolean
begin
    let s : integer = UInt(imms);
    let r : integer = UInt(immr);
    let width : integer = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && (immN::imms) != '1xxxxxx' then
        return FALSE;
    end;
    if sf == '0' && (immN::imms) != '00xxxxxx' then
        return FALSE;
    end;

    // for MOVZ must contain no more than 16 ones
    if s < 16 then
        // ones must not span halfword boundary when rotated
        return (-r MOD 16) <= (15 - s);
    end;

    // for MOVN must contain no more than 16 zeros
    if s >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (r MOD 16) <= (s - (width - 15));
    end;

    return FALSE;
end;
```

Library pseudocode for aarch64/functions/pac/addpac/AddPAC

```
// AddPAC()
// =====
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

func AddPAC(ptr : bits(64), modifier : bits(64), K : bits(128), data : boolean) => bits(64)
begin
    let use_modifier2 : boolean = FALSE;
    return InsertPAC(ptr, modifier, Zeros{64}, use_modifier2, K, data);
end;
```

Library pseudocode for aarch64/functions/pac/addpac/AddPAC2

```
// AddPAC2()
// =====
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

func AddPAC2(ptr : bits(64), modifier1 : bits(64), modifier2 : bits(64),
             K : bits(128), data : boolean) => bits(64)
begin
    let use_modifier2 : boolean = TRUE;
    return InsertPAC(ptr, modifier1, modifier2, use_modifier2, K, data);
end;
```



```

// InsertPAC()
// =====
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

func InsertPAC(ptr : bits(64), modifier : bits(64), modifier2 : bits(64), use_modifier2 : boolean,
               K : bits(128), data : boolean) => bits(64)
begin
    var PAC : bits(64);
    var result : bits(64);
    var ext_ptr : bits(64);
    var extfield : bits(64);
    var selbit : bit;
    let tbi : boolean = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    let mtx : boolean = (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) &&
                        EffectiveMTX(ptr, !data, PSTATE.EL) == '1');
    let top_bit : integer{} = if tbi then 55 else 63;
    let EL3_using_lva3 : boolean = (IsFeatureImplemented(FEAT_LVA3) &&
                                   TranslationRegime(PSTATE.EL) == Regime\_EL3 &&
                                   AArch64\_IASize(TCR\_EL3\(\).TOSZ) > 52);
    let is_VA_56bit : boolean = (TranslationRegime(PSTATE.EL) == Regime\_EL3 &&
                                 AArch64\_IASize(TCR\_EL3\(\).TOSZ) == 56);

    // If tagged pointers are in use for a regime with two TTBRs, use bit[55] of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of lxxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR\_EL1\(\).TBI1 == '1' || TCR\_EL1\(\).TBI0 == '1' then
                    selbit = ptr[55];
                else
                    selbit = ptr[63];
                end;
            else
                if ((TCR\_EL1\(\).TBI1 == '1' && TCR\_EL1\(\).TBID1 == '0') ||
                   (TCR\_EL1\(\).TBI0 == '1' && TCR\_EL1\(\).TBID0 == '0')) then
                    selbit = ptr[55];
                else
                    selbit = ptr[63];
                end;
            end;
        else
            // EL2 translation regime registers
            if data then
                if TCR\_EL2\(\).TBI1 == '1' || TCR\_EL2\(\).TBI0 == '1' then
                    selbit = ptr[55];
                else
                    selbit = ptr[63];
                end;
            else
                if ((TCR\_EL2\(\).TBI1 == '1' && TCR\_EL2\(\).TBID1 == '0') ||
                   (TCR\_EL2\(\).TBI0 == '1' && TCR\_EL2\(\).TBID0 == '0')) then
                    selbit = ptr[55];
                else
                    selbit = ptr[63];
                end;
            end;
        end;
    else
        selbit = if tbi then ptr[55] else ptr[63];
    end;

    if IsFeatureImplemented(FEAT_PAuth2) && IsFeatureImplemented(FEAT_CONSTPACFIELD) then
        selbit = ptr[55];
    end;
    let bottom_PAC_bit : AddressSize = CalculateBottomPACBit(selbit);

    if EL3_using_lva3 then
        extfield = Replicate{64}('0');
    else
        extfield = Replicate{64}(selbit);
    end;

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        if bottom_PAC_bit <= 55 then
            ext_ptr = (ptr[63:56] ::
                      extfield[55:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0]);
        else
            ext_ptr = ptr[63:56] :: ptr[55:0];
        end;
    elsif mtx then
        if bottom_PAC_bit <= 55 then
            ext_ptr = (extfield[63:60] :: ptr[59:56] ::
                      ptr[55:0]);
        else
            ext_ptr = (extfield[63:60] :: ptr[59:56] ::
                      ptr[55:0]);
        end;
    end;
end

```

```

        extfield[55:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0]);
    else
        ext_ptr = extfield[63:60] :: ptr[59:56] :: ptr[55:0];
    end;
else
    ext_ptr = extfield[63:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0];
end;

if use_modifier2 then
    assert IsFeatureImplemented(FEAT_PAuth_LR);
    PAC = ComputePAC2(ext_ptr, modifier, modifier2, K[127:64], K[63:0]);
else
    PAC = ComputePAC(ext_ptr, modifier, K[127:64], K[63:0]);
end;

if !IsFeatureImplemented(FEAT_PAuth2) then
    // If FEAT_PAuth2 is not implemented, the PAC is corrupted if the pointer does not have
    // a canonical VA.
    assert bottom_PAC_bit <= 52;
    if !IsZero(ptr[top_bit:bottom_PAC_bit]) && !IsOnes(ptr[top_bit:bottom_PAC_bit]) then
        PAC[top_bit-1] = NOT(PAC[top_bit-1]);
    end;
end;

// Preserve the determination between upper and lower address at bit[55] and insert PAC into
// bits that are not used for the address or the tag(s).
if !IsFeatureImplemented(FEAT_PAuth2) then
    assert (bottom_PAC_bit <= 52);
    if tbi then
        result = ptr[63:56]::selbit::PAC[54:bottom_PAC_bit]::ptr[bottom_PAC_bit-1:0];
    else
        result = PAC[63:56]::selbit::PAC[54:bottom_PAC_bit]::ptr[bottom_PAC_bit-1:0];
    end;
else
    var bit55 : bit;
    if EL3 using lva3 then
        // Bit 55 is an address bit (when VA size is 56-bits) or
        // used to store PAC (when VA size is less than 56-bits)
        if is_VA_56bit then
            bit55 = ptr[55];
        else
            bit55 = ptr[55] XOR PAC[55];
        end;
    else
        bit55 = selbit;
    end;
    if tbi then
        if bottom_PAC_bit < 55 then
            result = (ptr[63:56] :: bit55 ::
                (ptr[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit]) ::
                ptr[bottom_PAC_bit-1:0]);
        else
            result = (ptr[63:56] :: bit55 :: ptr[54:0]);
        end;
    elsif mtX then
        if bottom_PAC_bit < 55 then
            result = ((ptr[63:60] XOR PAC[63:60]) :: ptr[59:56] :: bit55 ::
                (ptr[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit]) ::
                ptr[bottom_PAC_bit-1:0]);
        else
            result = ((ptr[63:60] XOR PAC[63:60]) :: ptr[59:56] :: bit55 ::
                ptr[54:0]);
        end;
    else
        if bottom_PAC_bit < 55 then
            result = ((ptr[63:56] XOR PAC[63:56]) :: bit55 ::
                (ptr[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit]) ::
                ptr[bottom_PAC_bit-1:0]);
        else
            result = ((ptr[63:56] XOR PAC[63:56]) :: bit55 ::
                ptr[54:0]);
        end;
    end;
end;
return result;
end;

```

Library pseudocode for aarch64/functions/pac/addpacda/AddPACDA

```
// AddPACDA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDAKey_EL1.

func AddPACDA(x : bits(64), y : bits(64)) => bits(64)
begin
    let APDAKey_EL1 : bits(128) = APDAKeyHi_EL1()[63:0] :: APDAKeyLo_EL1()[63:0];
    if !IsAPDAKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APDAKey_EL1, TRUE);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/addpacdb/AddPACDB

```
// AddPACDB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDBKey_EL1.

func AddPACDB(x : bits(64), y : bits(64)) => bits(64)
begin
    let APDBKey_EL1 : bits(128) = APDBKeyHi_EL1()[63:0] :: APDBKeyLo_EL1()[63:0];
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

```
// AddPACGA()
// =====
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of x, y and the APGAKey_EL1.

func AddPACGA(x : bits(64), y : bits(64)) => bits(64)
begin
    var TrapEL2 : boolean;
    let APGAKey_EL1 : bits(128) = APGAKeyHi_EL1()[63:0] :: APGAKeyLo_EL1()[63:0];

    var TrapEL3 : boolean;
    case PSTATE.EL of
        when EL0 =>
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL3 =>
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
            TrapPACUse(EL3);
        end;
    else
        return ComputePAC(x, y, APGAKey_EL1[127:64], APGAKey_EL1[63:0])[63:32]::Zeros{32};
    end;
end;
```


Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA

```
// AddPACIA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, and the
// APIAKey_EL1.

func AddPACIA(x : bits(64), y : bits(64)) => bits(64)
begin
    let APIAKey_EL1 : bits(128) = APIAKeyHi_EL1()[63:0]::APIAKeyLo_EL1()[63:0];
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APIAKey_EL1, FALSE);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA2

```
// AddPACIA2()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, z, and
// the APIAKey_EL1.

func AddPACIA2(x : bits(64), y : bits(64), z : bits(64)) => bits(64)
begin
    let APIAKey_EL1 : bits(128) = APIAKeyHi_EL1()[63:0]::APIAKeyLo_EL1()[63:0];
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return AddPAC2(x, y, z, APIAKey_EL1, FALSE);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

```
// AddPACIB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APIBKey_EL1.

func AddPACIB(x : bits(64), y : bits(64)) => bits(64)
begin
    let APIBKey_EL1 : bits(128) = APIBKeyHi_EL1()[63:0] :: APIBKeyLo_EL1()[63:0];
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB2

```
// AddPACIB2()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, z, and
// the APIBKey_EL1.

func AddPACIB2(x : bits(64), y : bits(64), z : bits(64)) => bits(64)
begin
    let APIBKey_EL1 : bits(128) = APIBKeyHi_EL1()[63:0] :: APIBKeyLo_EL1()[63:0];
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return AddPAC2(x, y, z, APIBKey_EL1, FALSE);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/auth/AArch64_PACFailException

```
// AArch64_PACFailException()
// =====
// Generates a PAC Fail Exception

func AArch64_PACFailException(syndrome : bits(2))
begin
    let route_to_el2 : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2().TGE == '1';
    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_PACFail);
    except.syndrome.iss[1:0] = syndrome;
    except.syndrome.iss[24:2] = Zeros{23}; // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64_TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64_TakeException(EL1, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/auth/Auth

```
// Auth()
// =====
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

func Auth(ptr : bits(64), modifier : bits(64), K : bits(128), data : boolean, key_number : bit,
         is_combined : boolean) => bits(64)
begin
    let use_modifier2 : boolean = FALSE;
    return Authenticate(ptr, modifier, Zeros{64}, use_modifier2,
                       K, data, key_number, is_combined);
end;
```

Library pseudocode for aarch64/functions/pac/auth/Auth2

```
// Auth2()
// =====
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

func Auth2(ptr : bits(64), modifier1 : bits(64), modifier2 : bits(64), K : bits(128),
          data : boolean, key_number : bit, is_combined : boolean) => bits(64)
begin
    let use_modifier2 : boolean = TRUE;
    return Authenticate(ptr, modifier1, modifier2, use_modifier2, K, data, key_number, is_combined);
end;
```



```

// Authenticate()
// =====
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

func Authenticate(ptr : bits(64), modifier : bits(64), modifier2 : bits(64),
    use_modifier2 : boolean, K : bits(128), data : boolean, key_number : bit,
    is_combined : boolean) => bits(64)
begin
    var PAC : bits(64);
    var result : bits(64);
    var original_ptr : bits(64);
    var error_code : bits(2);
    var extfield : bits(64);

    // Reconstruct the extension field used of adding the PAC to the pointer
    let tbi : boolean = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    let mtx : boolean = (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) &&
        EffectiveMTX(ptr, !data, PSTATE.EL) == '1');
    let bottom_PAC_bit : AddressSize = CalculateBottomPACBit(ptr[55]);
    let EL3_using_lva3 : boolean = (IsFeatureImplemented(FEAT_LVA3) &&
        TranslationRegime(PSTATE.EL) == Regime\_EL3 &&
        AArch64\_IASize(TCR\_EL3\(\).T0SZ) > 52);
    let is_VA_56bit : boolean = (TranslationRegime(PSTATE.EL) == Regime\_EL3 &&
        AArch64\_IASize(TCR\_EL3\(\).T0SZ) == 56);

    if EL3_using_lva3 then
        extfield = Replicate{64}('0');
    else
        extfield = Replicate{64}(ptr[55]);
    end;

    if tbi then
        if bottom_PAC_bit <= 55 then
            original_ptr = (ptr[63:56] ::
                extfield[55:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0]);
        else
            original_ptr = ptr[63:56] :: ptr[55:0];
        end;
    elseif mtx then
        if bottom_PAC_bit <= 55 then
            original_ptr = (extfield[63:60] :: ptr[59:56] ::
                extfield[55:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0]);
        else
            original_ptr = extfield[63:60] :: ptr[59:56] :: ptr[55:0];
        end;
    else
        original_ptr = extfield[63:bottom_PAC_bit] :: ptr[bottom_PAC_bit-1:0];
    end;

    if use_modifier2 then
        assert IsFeatureImplemented(FEAT_PAuth_LR);
        PAC = ComputePAC2(original_ptr, modifier, modifier2, K[127:64], K[63:0]);
    else
        PAC = ComputePAC(original_ptr, modifier, K[127:64], K[63:0]);
    end;

    // Check pointer authentication code
    if tbi then
        if !IsFeatureImplemented(FEAT_PAuth2) then
            assert (bottom_PAC_bit <= 52);
            if PAC[54:bottom_PAC_bit] == ptr[54:bottom_PAC_bit] then
                result = original_ptr;
            else
                error_code = key_number::NOT(key_number);
                result = original_ptr[63:55]::error_code::original_ptr[52:0];
            end;
        else
            result = ptr;
            if EL3_using_lva3 && !is_VA_56bit then
                result[55] = result[55] XOR PAC[55];
            end;
            if (bottom_PAC_bit < 55) then
                result[54:bottom_PAC_bit] = result[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit];
            end;
            if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
                (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
                if (EL3_using_lva3 && !is_VA_56bit && !IsZero(result[55:bottom_PAC_bit])) then
                    error_code = (if data then '1' else '0')::key_number;
                    AArch64\_PACFailException(error_code);
                elseif (!EL3_using_lva3 && (bottom_PAC_bit < 55) &&
                    result[54:bottom_PAC_bit] !=
                    Replicate{55-bottom_PAC_bit}(result[55])) then
                    error_code = (if data then '1' else '0')::key_number;
                    AArch64\_PACFailException(error_code);
                end;
            end;
        end;
    end;
end;

```

```

elseif mtX then
    assert IsFeatureImplemented(FEAT_PAuth2);
    result = ptr;
    if EL3_using_lva3 && !is_VA_56bit then
        result[55] = result[55] XOR PAC[55];
    end;
    if (bottom_PAC_bit < 55) then
        result[54:bottom_PAC_bit] = result[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit];
    end;
    result[63:60] = result[63:60] XOR PAC[63:60];
    if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
        (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
        if (EL3_using_lva3 && !is_VA_56bit &&
            (!IsZero(result[55:bottom_PAC_bit]) || !IsZero(result[63:60]))) then
            error_code = (if data then '1' else '0')::key_number;
            AArch64\_PACFailException(error_code);
        elseif (!EL3_using_lva3 && (bottom_PAC_bit < 55) &&
            ((result[54:bottom_PAC_bit] !=
                Replicate{55-bottom_PAC_bit}(result[55])) ||
            (result[63:60] != Replicate{4}(result[55])))) then
            error_code = (if data then '1' else '0')::key_number;
            AArch64\_PACFailException(error_code);
        end;
    end;
else
    if !IsFeatureImplemented(FEAT_PAuth2) then
        assert (bottom_PAC_bit <= 52);
        if PAC[54:bottom_PAC_bit] == ptr[54:bottom_PAC_bit] && PAC[63:56] == ptr[63:56] then
            result = original_ptr;
        else
            error_code = key_number::NOT(key_number);
            result = original_ptr[63]::error_code::original_ptr[60:0];
        end;
    else
        result = ptr;
        if EL3_using_lva3 && !is_VA_56bit then
            result[55] = result[55] XOR PAC[55];
        end;
        if bottom_PAC_bit < 55 then
            result[54:bottom_PAC_bit] = result[54:bottom_PAC_bit] XOR PAC[54:bottom_PAC_bit];
        end;
        result[63:56] = result[63:56] XOR PAC[63:56];
        if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
            (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
            if (EL3_using_lva3 && !IsZero(result[63:bottom_PAC_bit])) then
                error_code = (if data then '1' else '0')::key_number;
                AArch64\_PACFailException(error_code);
            elseif (!EL3_using_lva3 &&
                result[63:bottom_PAC_bit] !=
                Replicate{64-bottom_PAC_bit}(result[55])) then
                error_code = (if data then '1' else '0')::key_number;
                AArch64\_PACFailException(error_code);
            end;
        end;
    end;
end;
return result;
end;

```

Library pseudocode for aarch64/functions/pac/authda/AuthDA

```

// AuthDA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACDA().

func AuthDA(x : bits(64), y : bits(64), is_combined : boolean) => bits(64)
begin
    let APDAKey_EL1 : bits(128) = APDAKeyHi_EL1()[63:0] :: APDAKeyLo_EL1()[63:0];
    if !IsAPDAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDAKey_EL1, TRUE, '0', is_combined);
    end;
end;

```

Library pseudocode for aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of x, using
// the same algorithm and key as AddPACDB().

func AuthDB(x : bits(64), y : bits(64), is_combined : boolean) => bits(64)
begin
    let APDBKey_EL1 : bits(128) = APDBKeyHi_EL1()[63:0] :: APDBKeyLo_EL1()[63:0];
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA().

func AuthIA(x : bits(64), y : bits(64), is_combined : boolean) => bits(64)
begin
    let APIAKey_EL1 : bits(128) = APIAKeyHi_EL1()[63:0] :: APIAKeyLo_EL1()[63:0];
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIAKey_EL1, FALSE, '0', is_combined);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/authia/AuthIA2

```
// AuthIA2()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA2().

func AuthIA2(x : bits(64), y : bits(64), z : bits(64), is_combined : boolean) => bits(64)
begin
    let APIAKey_EL1 : bits(128) = APIAKeyHi_EL1()[63:0] :: APIAKeyLo_EL1()[63:0];
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return Auth2(x, y, z, APIAKey_EL1, FALSE, '0', is_combined);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB().

func AuthIB(x : bits(64), y : bits(64), is_combined : boolean) => bits(64)
begin
    let APIBKey_EL1 : bits(128) = APIBKeyHi_EL1()[63:0] :: APIBKeyLo_EL1()[63:0];
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/authib/AuthIB2

```
// AuthIB2()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB2().

func AuthIB2(x : bits(64), y : bits(64), z : bits(64), is_combined : boolean) => bits(64)
begin
    let APIBKey_EL1 : bits(128) = APIBKeyHi_EL1()[63:0] :: APIBKeyLo_EL1()[63:0];
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return Auth2(x, y, z, APIBKey_EL1, FALSE, '1', is_combined);
    end;
end;
```

Library pseudocode for aarch64/functions/pac/calcbottompacbit/AArch64_PACEffectiveTxSZ

```
// AArch64_PACEffectiveTxSZ()
// =====
// Compute the effective value for TxSZ used to determine the placement of the PAC field

func AArch64_PACEffectiveTxSZ(regime : Regime, walkparams : S1TTWParams) => bits(6)
begin
    let slmaxtxsz : integer = AArch64_MaxTxSZ(walkparams.tgx);
    let slmintxsz : integer = AArch64_S1MinTxSZ(regime, walkparams);

    if AArch64_S1TxSZFaults(regime, walkparams) then
        if ConstrainUnpredictable(Unpredictable_RESTnSZ) == Constraint_FORCE then
            if UInt(walkparams.txsz) < slmintxsz then
                return slmintxsz[5:0];
            end;
            if UInt(walkparams.txsz) > slmaxtxsz then
                return slmaxtxsz[5:0];
            end;
        end;
    elseif UInt(walkparams.txsz) < slmintxsz then
        return slmintxsz[5:0];
    elseif UInt(walkparams.txsz) > slmaxtxsz then
        return slmaxtxsz[5:0];
    end;

    return walkparams.txsz;
end;
```

Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =====

func CalculateBottomPACBit(top_bit : bit) => AddressSize
begin
    var regime : Regime;
    var walkparams : S1TTWParams;
    var bottom_PAC_bit : AddressSize;

    regime = TranslationRegime(PSTATE.EL);
    let ss : SecurityState = CurrentSecurityState();
    walkparams = AArch64_GetS1TTWParams(regime, PSTATE.EL, ss, Replicate{64}(top_bit));
    bottom_PAC_bit = ((64 - UInt(AArch64_PACEffectiveTxSZ(regime, walkparams)))
        as AddressSize);

    return bottom_PAC_bit;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```
// ComputePAC()
// =====

func ComputePAC(data : bits(64), modifier : bits(64), key0 : bits(64), key1 : bits(64)) => bits(64)
begin
    if IsFeatureImplemented(FEAT_PACIMP) then
        return ComputePACIMPDEF(data, modifier, key0, key1);
    end;
    if IsFeatureImplemented(FEAT_PACQARMA3) then
        let isqarma3 : boolean = TRUE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    end;
    if IsFeatureImplemented(FEAT_PACQARMA5) then
        let isqarma3 : boolean = FALSE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    end;
    unreachable;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePAC2

```
// ComputePAC2()
// =====

func ComputePAC2(data : bits(64), modifier1 : bits(64), modifier2 : bits(64),
    key0 : bits(64), key1 : bits(64)) => bits(64)
begin
    if IsFeatureImplemented(FEAT_PACIMP) then
        return ComputePAC2IMPDEF(data, modifier1, modifier2, key0, key1);
    end;
    if IsFeatureImplemented(FEAT_PACQARMA3) then
        let isqarma3 : boolean = TRUE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    end;
    if IsFeatureImplemented(FEAT_PACQARMA5) then
        let isqarma3 : boolean = FALSE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    end;
    unreachable;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePAC2IMPDEF

```
// ComputePAC2IMPDEF()
// =====
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

impdef func ComputePAC2IMPDEF(data : bits(64),
    modifier1 : bits(64),
    modifier2 : bits(64),
    key0 : bits(64),
    key1 : bits(64)) => bits(64)
begin
    Unimplemented();
end;
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePAC2QARMA

```
// ComputePAC2QARMA()
// =====

func ComputePAC2QARMA(data : bits(64),
    modifier1 : bits(64),
    modifier2 : bits(64),
    key0 : bits(64),
    key1 : bits(64),
    isqarma3 : boolean) => bits(64)
begin
    let concat_modifiers : bits(64) = modifier2[36:5]::modifier1[35:4];
    return ComputePACQARMA(data, concat_modifiers, key0, key1, isqarma3);
end;
```


Library pseudocode for aarch64/functions/pac/computepac/ComputePACIMPDEF

```
// ComputePACIMPDEF()
// =====
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

impdef func ComputePACIMPDEF(data : bits(64), modifier : bits(64),
                             key0 : bits(64), key1 : bits(64)) => bits(64)
begin
    Unimplemented();
end;
```



```

// ComputePACQARMA()
// =====
// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation

func ComputePACQARMA(data : bits(64),
    modifier : bits(64),
    key0 : bits(64),
    key1 : bits(64),
    isqarma3 : boolean) => bits(64)
begin
    var workingval : bits(64);
    var runningmod : bits(64);
    var roundkey : bits(64);
    var modk0 : bits(64);
    let Alpha : bits(64) = 0xC0AC29B7C97C50DD[63:0];

    var iterations : integer;
    RC[[0]] = 0x0000000000000000[63:0];
    RC[[1]] = 0x13198A2E03707344[63:0];
    RC[[2]] = 0xA4093822299F31D0[63:0];

    if isqarma3 then
        iterations = 2;
    else // QARMA5
        iterations = 4;
        RC[[3]] = 0x082EFA98EC4E6C89[63:0];
        RC[[4]] = 0x452821E638D01377[63:0];
    end;

    modk0 = key0[0]::key0[63:2]::(key0[63] XOR key0[1]);
    runningmod = modifier;
    workingval = data XOR key0;

    for i = 0 to iterations do
        roundkey = key1 XOR runningmod;
        workingval = workingval XOR roundkey;
        workingval = workingval XOR RC[[i]];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        end;
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACSub(workingval);
        end;
        runningmod = TweakShuffle(runningmod[63:0]);
    end;
    roundkey = modk0 XOR runningmod;
    workingval = workingval XOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    end;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 XOR workingval;
    workingval = PACCellInvShuffle(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    end;
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval XOR key0;
    workingval = workingval XOR runningmod;
    for i = 0 to iterations do
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACInvSub(workingval);
        end;
        if i < iterations then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
        end;
        runningmod = TweakInvShuffle(runningmod[63:0]);
        roundkey = key1 XOR runningmod;
        workingval = workingval XOR RC[[iterations-i]];
        workingval = workingval XOR roundkey;
        workingval = workingval XOR Alpha;
    end;
    workingval = workingval XOR modk0;

```

```

    return workingval;
end;

```

Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```

// PACCellInvShuffle()
// =====

func PACCellInvShuffle(indata : bits(64)) => bits(64)
begin
    var outdata : bits(64);
    outdata[3:0]   = indata[15:12];
    outdata[7:4]   = indata[27:24];
    outdata[11:8]  = indata[51:48];
    outdata[15:12] = indata[39:36];
    outdata[19:16] = indata[59:56];
    outdata[23:20] = indata[47:44];
    outdata[27:24] = indata[7:4];
    outdata[31:28] = indata[19:16];
    outdata[35:32] = indata[35:32];
    outdata[39:36] = indata[55:52];
    outdata[43:40] = indata[31:28];
    outdata[47:44] = indata[11:8];
    outdata[51:48] = indata[23:20];
    outdata[55:52] = indata[3:0];
    outdata[59:56] = indata[43:40];
    outdata[63:60] = indata[63:60];
    return outdata;
end;

```

Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```

// PACCellShuffle()
// =====

func PACCellShuffle(indata : bits(64)) => bits(64)
begin
    var outdata : bits(64);
    outdata[3:0]   = indata[55:52];
    outdata[7:4]   = indata[27:24];
    outdata[11:8]  = indata[47:44];
    outdata[15:12] = indata[3:0];
    outdata[19:16] = indata[31:28];
    outdata[23:20] = indata[51:48];
    outdata[27:24] = indata[7:4];
    outdata[31:28] = indata[43:40];
    outdata[35:32] = indata[35:32];
    outdata[39:36] = indata[15:12];
    outdata[43:40] = indata[59:56];
    outdata[47:44] = indata[23:20];
    outdata[51:48] = indata[11:8];
    outdata[55:52] = indata[39:36];
    outdata[59:56] = indata[19:16];
    outdata[63:60] = indata[63:60];
    return outdata;
end;

```

Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// =====

func PACInvSub(Tinput : bits(64)) => bits(64)
begin
    // This is a 4-bit substitution from the PRINCE-family cipher
    var Toutput : bits(64);
    for i = 0 to 15 do
        case Tinput[i*:4] of
            when '0000' => Toutput[i*:4] = '0101';
            when '0001' => Toutput[i*:4] = '1110';
            when '0010' => Toutput[i*:4] = '1101';
            when '0011' => Toutput[i*:4] = '1000';
            when '0100' => Toutput[i*:4] = '1010';
            when '0101' => Toutput[i*:4] = '1011';
            when '0110' => Toutput[i*:4] = '0001';
            when '0111' => Toutput[i*:4] = '1001';
            when '1000' => Toutput[i*:4] = '0010';
            when '1001' => Toutput[i*:4] = '0110';
            when '1010' => Toutput[i*:4] = '1111';
            when '1011' => Toutput[i*:4] = '0000';
            when '1100' => Toutput[i*:4] = '0100';
            when '1101' => Toutput[i*:4] = '1100';
            when '1110' => Toutput[i*:4] = '0111';
            when '1111' => Toutput[i*:4] = '0011';
        end;
    end;
    return Toutput;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =====

func PACMult(Sinput : bits(64)) => bits(64)
begin
    var t0 : bits(4);
    var t1 : bits(4);
    var t2 : bits(4);
    var t3 : bits(4);
    var Soutput : bits(64);

    for i = 0 to 3 do
        t0[3:0] = ROL(Sinput[((i+8))*:4], 1) XOR ROL(Sinput[((i+4))*:4], 2);
        t0[3:0] = t0[3:0] XOR ROL(Sinput[i*:4], 1);
        t1[3:0] = ROL(Sinput[((i+12))*:4], 1) XOR ROL(Sinput[((i+4))*:4], 1);
        t1[3:0] = t1[3:0] XOR ROL(Sinput[i*:4], 2);
        t2[3:0] = ROL(Sinput[((i+12))*:4], 2) XOR ROL(Sinput[((i+8))*:4], 1);
        t2[3:0] = t2[3:0] XOR ROL(Sinput[i*:4], 1);
        t3[3:0] = ROL(Sinput[((i+12))*:4], 1) XOR ROL(Sinput[((i+8))*:4], 2);
        t3[3:0] = t3[3:0] XOR ROL(Sinput[((i+4))*:4], 1);
        Soutput[i*:4] = t3[3:0];
        Soutput[((i+4))*:4] = t2[3:0];
        Soutput[((i+8))*:4] = t1[3:0];
        Soutput[((i+12))*:4] = t0[3:0];
    end;
    return Soutput;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// =====

func PACSub(Tinput : bits(64)) => bits(64)
begin
    // This is a 4-bit substitution from the PRINCE-family cipher
    var Toutput : bits(64);
    for i = 0 to 15 do
        case Tinput[i*4] of
            when '0000' => Toutput[i*4] = '1011';
            when '0001' => Toutput[i*4] = '0110';
            when '0010' => Toutput[i*4] = '1000';
            when '0011' => Toutput[i*4] = '1111';
            when '0100' => Toutput[i*4] = '1100';
            when '0101' => Toutput[i*4] = '0000';
            when '0110' => Toutput[i*4] = '1001';
            when '0111' => Toutput[i*4] = '1110';
            when '1000' => Toutput[i*4] = '0011';
            when '1001' => Toutput[i*4] = '0111';
            when '1010' => Toutput[i*4] = '0100';
            when '1011' => Toutput[i*4] = '0101';
            when '1100' => Toutput[i*4] = '1101';
            when '1101' => Toutput[i*4] = '0010';
            when '1110' => Toutput[i*4] = '0001';
            when '1111' => Toutput[i*4] = '1010';
        end;
    end;
    return Toutput;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/PacSub1

```
// PacSub1()
// =====

func PACSub1(Tinput : bits(64)) => bits(64)
begin
    // This is a 4-bit substitution from Qarma signal
    var Toutput : bits(64);
    for i = 0 to 15 do
        case Tinput[i*4] of
            when '0000' => Toutput[i*4] = '1010';
            when '0001' => Toutput[i*4] = '1101';
            when '0010' => Toutput[i*4] = '1110';
            when '0011' => Toutput[i*4] = '0110';
            when '0100' => Toutput[i*4] = '1111';
            when '0101' => Toutput[i*4] = '0111';
            when '0110' => Toutput[i*4] = '0011';
            when '0111' => Toutput[i*4] = '0101';
            when '1000' => Toutput[i*4] = '1001';
            when '1001' => Toutput[i*4] = '1000';
            when '1010' => Toutput[i*4] = '0000';
            when '1011' => Toutput[i*4] = '1100';
            when '1100' => Toutput[i*4] = '1011';
            when '1101' => Toutput[i*4] = '0001';
            when '1110' => Toutput[i*4] = '0010';
            when '1111' => Toutput[i*4] = '0100';
        end;
    end;
    return Toutput;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/RC

```
// RC[]
// ====

var RC : array [[5]] of bits(64);
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =====

func TweakCellInvRot(incell : bits(4)) => bits(4)
begin
    var outcell : bits(4);
    outcell[3] = incell[2];
    outcell[2] = incell[1];
    outcell[1] = incell[0];
    outcell[0] = incell[0] XOR incell[3];
    return outcell;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// =====

func TweakCellRot(incell : bits(4)) => bits(4)
begin
    var outcell : bits(4);
    outcell[3] = incell[0] XOR incell[1];
    outcell[2] = incell[3];
    outcell[1] = incell[2];
    outcell[0] = incell[1];
    return outcell;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =====

func TweakInvShuffle(indata : bits(64)) => bits(64)
begin
    var outdata : bits(64);
    outdata[3:0]   = TweakCellInvRot(indata[51:48]);
    outdata[7:4]   = indata[55:52];
    outdata[11:8]  = indata[23:20];
    outdata[15:12] = indata[27:24];
    outdata[19:16] = indata[3:0];
    outdata[23:20] = indata[7:4];
    outdata[27:24] = TweakCellInvRot(indata[11:8]);
    outdata[31:28] = indata[15:12];
    outdata[35:32] = TweakCellInvRot(indata[31:28]);
    outdata[39:36] = TweakCellInvRot(indata[63:60]);
    outdata[43:40] = TweakCellInvRot(indata[59:56]);
    outdata[47:44] = TweakCellInvRot(indata[19:16]);
    outdata[51:48] = indata[35:32];
    outdata[55:52] = indata[39:36];
    outdata[59:56] = indata[43:40];
    outdata[63:60] = TweakCellInvRot(indata[47:44]);
    return outdata;
end;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// =====

func TweakShuffle(indata : bits(64)) => bits(64)
begin
    var outdata : bits(64);
    outdata[3:0]   = indata[19:16];
    outdata[7:4]   = indata[23:20];
    outdata[11:8]  = TweakCellRot(indata[27:24]);
    outdata[15:12] = indata[31:28];
    outdata[19:16] = TweakCellRot(indata[47:44]);
    outdata[23:20] = indata[11:8];
    outdata[27:24] = indata[15:12];
    outdata[31:28] = TweakCellRot(indata[35:32]);
    outdata[35:32] = indata[51:48];
    outdata[39:36] = indata[55:52];
    outdata[43:40] = indata[59:56];
    outdata[47:44] = TweakCellRot(indata[63:60]);
    outdata[51:48] = TweakCellRot(indata[3:0]);
    outdata[55:52] = indata[7:4];
    outdata[59:56] = TweakCellRot(indata[43:40]);
    outdata[63:60] = TweakCellRot(indata[39:36]);
    return outdata;
end;
```

Library pseudocode for aarch64/functions/pac/pac/IsAPDAKeyEnabled

```
// IsAPDAKeyEnabled()
// =====
// Returns TRUE if authentication using the APDAKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

func IsAPDAKeyEnabled() => boolean
begin
    var TrapEL2 : boolean;
    var TrapEL3 : boolean;
    var Enable : bits(1);

    case PSTATE.EL of
        when EL0 =>
            let IsEL1Regime : boolean = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1().EnDA else SCTLRL_EL2().EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            Enable = SCTLRL_EL1().EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            Enable = SCTLRL_EL2().EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL3 =>
            Enable = SCTLRL_EL3().EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
            TrapPACUse(EL3);
        end;
    else
        return TRUE;
    end;
end;
```


Library pseudocode for aarch64/functions/pac/pac/IsAPDBKeyEnabled

```
// IsAPDBKeyEnabled()
// =====
// Returns TRUE if authentication using the APDBKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

func IsAPDBKeyEnabled() => boolean
begin
    var TrapEL2 : boolean;
    var TrapEL3 : boolean;
    var Enable : bits(1);

    case PSTATE.EL of
        when EL0 =>
            let IsEL1Regime : boolean = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1().EnDB else SCTLRL_EL2().EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            Enable = SCTLRL_EL1().EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            Enable = SCTLRL_EL2().EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL3 =>
            Enable = SCTLRL_EL3().EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
            TrapPACUse(EL3);
        end;
    else
        return TRUE;
    end;
end;
```

Library pseudocode for aarch64/functions/pac/pac/IsAPIAKeyEnabled

```
// IsAPIAKeyEnabled()
// =====
// Returns TRUE if authentication using the APIAKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

func IsAPIAKeyEnabled() => boolean
begin
    var TrapEL2 : boolean;
    var TrapEL3 : boolean;
    var Enable : bits(1);

    case PSTATE.EL of
        when EL0 =>
            let IsEL1Regime : boolean = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1().EnIA else SCTLRL_EL2().EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            Enable = SCTLRL_EL1().EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            Enable = SCTLRL_EL2().EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL3 =>
            Enable = SCTLRL_EL3().EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
            TrapPACUse(EL3);
        end;
    else
        return TRUE;
    end;
end;
```

Library pseudocode for aarch64/functions/pac/pac/IsAPIBKeyEnabled

```
// IsAPIBKeyEnabled()
// =====
// Returns TRUE if authentication using the APIBKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

func IsAPIBKeyEnabled() => boolean
begin
    var TrapEL2 : boolean;
    var TrapEL3 : boolean;
    var Enable : bits(1);

    case PSTATE.EL of
        when EL0 =>
            let IsEL1Regime : boolean = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1().EnIB else SCTL_EL2().EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL1 =>
            Enable = SCTL_EL1().EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2().API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL2 =>
            Enable = SCTL_EL2().EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3().API == '0';
        when EL3 =>
            Enable = SCTL_EL3().EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        Undefined();
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            Undefined();
        else
            TrapPACUse(EL3);
        end;
    else
        return TRUE;
    end;
end;
```

Library pseudocode for aarch64/functions/pac/pac/IsPACMEnabled

```
// IsPACMEnabled()
// =====
// Returns TRUE if the effects of the PACM instruction are enabled, otherwise FALSE.

func IsPACMEnabled() => boolean
begin
    assert IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_PAuth_LR);

    if IsTrivialPACMImplementation() then
        return FALSE;
    end;

    var enabled : boolean;

    // EL2 could force the behavior at EL1 and EL0 to NOP.
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        enabled = IsHCRXEL2Enabled() && HCRX\_EL2().PACMEN == '1';
    else
        enabled = TRUE;
    end;

    // Otherwise, the SCTLR2_ELx bit determines the behavior.
    if enabled then
        var enpacm_bit : bit;
        case PSTATE.EL of
            when EL3 =>
                enpacm_bit = SCTLR2\_EL3().EnPACM;
            when EL2 =>
                enpacm_bit = if IsSCTLR2EL2Enabled() then SCTLR2\_EL2().EnPACM else '0';
            when EL1 =>
                enpacm_bit = if IsSCTLR2EL1Enabled() then SCTLR2\_EL1().EnPACM else '0';
            when EL0 =>
                if IsInHost() then
                    enpacm_bit = if IsSCTLR2EL2Enabled() then SCTLR2\_EL2().EnPACM0 else '0';
                else
                    enpacm_bit = if IsSCTLR2EL1Enabled() then SCTLR2\_EL1().EnPACM0 else '0';
                end;
        end;
        enabled = enpacm_bit == '1';
    end;

    return enabled;
end;
```

Library pseudocode for aarch64/functions/pac/pac/IsTrivialPACMImplementation

```
// IsTrivialPACMImplementation()
// =====
// Returns TRUE if the PE has a trivial implementation of PACM.

func IsTrivialPACMImplementation() => boolean
begin
    return (IsFeatureImplemented(FEAT_PACIMP) &&
        ImpDefBool("Trivial PSTATE.PACM implementation"));
end;
```

Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// =====
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

func PtrHasUpperAndLowerAddRanges() => boolean
begin
    let regime : Regime = TranslationRegime(PSTATE.EL);

    return HasUnprivileged(regime);
end;
```

Library pseudocode for aarch64/functions/pac/strip/Strip

```
// Strip()
// =====
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

func Strip(A : bits(64), data : boolean) => bits(64)
begin
    var original_ptr : bits(64);
    var extfield : bits(64);
    let tbi : boolean = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    let mtX : boolean = (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) &&
        EffectiveMTX(A, !data, PSTATE.EL) == '1');
    let bottom_PAC_bit : AddressSize = CalculateBottomPACBit(A[55]);
    let EL3_using_lva3 : boolean = (IsFeatureImplemented(FEAT_LVA3) &&
        TranslationRegime(PSTATE.EL) == Regime\_EL3 &&
        AArch64\_IASize(TCR_EL3().T0SZ) > 52);

    if EL3_using_lva3 then
        extfield = Replicate{64}('0');
    else
        extfield = Replicate{64}(A[55]);
    end;

    if tbi then
        if (bottom_PAC_bit <= 55) then
            original_ptr = (A[63:56] ::
                extfield[55:bottom_PAC_bit] :: A[bottom_PAC_bit-1:0]);

            else
                original_ptr = A[63:56] :: A[55:0];
            end;
        elseif mtX then
            if (bottom_PAC_bit <= 55) then
                original_ptr = (extfield[63:60] :: A[59:56] ::
                    extfield[55:bottom_PAC_bit] :: A[bottom_PAC_bit-1:0]);

                else
                    original_ptr = extfield[63:60] :: A[59:56] :: A[55:0];
                end;
            else
                original_ptr = extfield[63:bottom_PAC_bit] :: A[bottom_PAC_bit-1:0];
            end;

        return original_ptr;
    end;
```

Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// =====
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

noreturn func TrapPACUse(target_el : bits(2))
begin
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    let preferred_exception_return : bits(64) = ThisInstrAddr{}();
    var except : ExceptionRecord;
    let vect_offset : integer = 0;
    except = ExceptionSyndrome(Exception\_PACTrap);
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
end;
```

Library pseudocode for aarch64/functions/predictionrestrict/AArch64_RestrictPrediction

```
// AArch64_RestrictPrediction()
// =====
// Clear all predictions in the context.

func AArch64_RestrictPrediction(val : bits(64), restriction : RestrictType)
begin
    var c : ExecutionCntxt;
    let target_el : bits(2) = val[25:24];

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then ExecuteAsNOP(); end;

    let ns : bit = val[26];
    let nse : bit = val[27];
    let ss : SecurityState = TargetSecurityState(ns, nse);

    // If the combination of Security state and Exception level is not implemented,
    // the instruction is treated as a NOP.
    if ss == SS_Root && target_el != EL3 then ExecuteAsNOP(); end;
    if !IsFeatureImplemented(FEAT_RME) && target_el == EL3 && ss != SS_Secure then
        ExecuteAsNOP();
    end;

    c.security = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if (PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1 then
            c.is_vmid_valid = TRUE;
            c.all_vmid = FALSE;
            c.vmid = VMID();

            elsif (target_el == EL0 && !ELIsInHost(target_el)) || target_el == EL1 then
                c.is_vmid_valid = TRUE;
                c.all_vmid = val[48] == '1';
                c.vmid = val[47:32]; // Only valid if val[48] == '0';

            else
                c.is_vmid_valid = FALSE;
            end;
        else
            c.is_vmid_valid = FALSE;
        end;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid = FALSE;
        c.asid = ASID();

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid = val[16] == '1';
        c.asid = val[15:0]; // Only valid if val[16] == '0';

    else
        c.is_asid_valid = FALSE;
    end;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
end;
```

Library pseudocode for aarch64/functions/prefetch/Prefetch

```
// Prefetch()
// =====

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

func Prefetch(address : bits(64), prfop : bits(5), supports_ir : boolean)
begin
    var hint : PrefetchHint;
    var target : integer;
    var stream : boolean;

    case prfop of
        when '00xxx' => hint = Prefetch READ; // PLD: prefetch for load
        when '01xxx' => hint = Prefetch EXEC; // PLI: preload instructions
        when '10xxx' => hint = Prefetch WRITE; // PST: prepare for store
        when '11000' where supports_ir => hint = Prefetch IR; // IR: intent to read
        otherwise => return; // unallocated hint
    end;

    // If <type> is IR, then <target> and <policy> are not specified and Rt[2:0] are '000'.
    if hint != Prefetch IR then
        target = UInt(prfop[2:1]); // target cache level
        stream = (prfop[0] != '0'); // streaming (non-temporal)
    else
        target = ARBITRARY: integer;
        stream = ARBITRARY: boolean;
    end;
    Hint_Prefetch(address, hint, target, stream);
    return;
end;
```

Library pseudocode for aarch64/functions/pstatefield/PSTATEField

```
// PSTATEField
// =====
// MSR (immediate) instruction destinations.

type PSTATEField of enumeration {
    PSTATEField_DAIFSet, PSTATEField_DAIFClr,
    PSTATEField_PAN, // Armv8.1
    PSTATEField_UAO, // Armv8.2
    PSTATEField_DIT, // Armv8.4
    PSTATEField_SSBS,
    PSTATEField_TCO, // Armv8.5
    PSTATEField_SVCRSM,
    PSTATEField_SVCRZA,
    PSTATEField_SVCRSMZA,
    PSTATEField_ALLINT,
    PSTATEField_PM,
    PSTATEField_SP
};
```

Library pseudocode for aarch64/functions/ras/AArch64_DelegatedSErrorTarget

```
// AArch64_DelegatedSErrorTarget()
// =====
// Returns whether a delegated SError exception pended by SCR_EL3.VSE is masked,
// and the target Exception level of the delegated SError exception.

func AArch64_DelegatedSErrorTarget() => (boolean, bits(2))
begin
    assert IsFeatureImplemented(FEAT_E3DSE);
    if Halted() || PSTATE.EL == EL3 then
        return (TRUE, ARBITRARY : bits(2));
    end;

    let effective_amo : bit = EffectiveHCR AMO();
    let effective_tge : bit = EffectiveTGE();

    // The exception is masked by software.
    var masked : boolean;
    case PSTATE.EL of
        when EL2 =>
            masked = ((effective_tge == '0' && effective_amo == '0') || PSTATE.A == '1');
        when EL1, EL0 =>
            masked = (effective_amo == '0' && PSTATE.A == '1');
        otherwise =>
            unreachable;
    end;

    // The exception might be disabled debug in the Security state indicated by
    // SCR_EL3().[NS, NSE] by external debug.
    let intdis : boolean = ExternalDebugInterruptsDisabled(EL1);

    var target_el : bits(2) = ARBITRARY : bits(2);
    if EL2Enabled() && effective_amo == '1' && !intdis && PSTATE.EL IN {EL0, EL1} then
        target_el = EL2;
        masked = FALSE;

    elseif (EffectiveHCRX EL2 TMEA() == '1' && !intdis &&
            ((PSTATE.EL == EL1 && PSTATE.A == '1') ||
             (PSTATE.EL == EL0 && masked && !IsInHost()))) then
        target_el = EL2;
        masked = FALSE;

    elseif PSTATE.EL == EL2 || IsInHost() then
        if !masked then target_el = EL2; end;

    else
        assert (PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()));
        if !masked then target_el = EL1; end;
    end;

    // External debug might disable the delegated exception for the target Exception level.
    if ExternalDebugInterruptsDisabled(target_el) then
        masked = TRUE;
        target_el = ARBITRARY : bits(2);
    end;

    return (masked, target_el);
end;
```


Library pseudocode for aarch64/functions/ras/AArch64_ESBOperation

```
// AArch64_ESBOperation()
// =====
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

func AArch64_ESBOperation()
begin
    var target_el : bits(2);
    var masked : boolean;

    (masked, target_el) = PhysicalErrorTarget();

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interprocessing case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            var syndrome : bits(32) = Zeros{};
            syndrome[31] = '1'; // A
            syndrome[15:0] = AArch32\_PhysicalErrorSyndrome();
            DISR() = syndrome;
        else
            let implicit_esb : boolean = FALSE;
            let is_esb : boolean = TRUE;
            var syndrome : bits(64) = Zeros{};

            syndrome[31] = '1'; // A
            syndrome[24:0] = AArch64\_PhysicalErrorSyndrome(is_esb, implicit_esb);
            DISR_EL1() = syndrome;
        end;
        ClearPendingPhysicalSError(); // Set ISR_EL1.A to 0
    end;

    return;
end;
```

Library pseudocode for aarch64/functions/ras/AArch64_EncodeAsyncErrorSyndrome

```
// AArch64_EncodeAsyncErrorSyndrome()
// =====
// Return the encoding for specified ErrorState for an SError exception taken
// to AArch64 state.

func AArch64_EncodeAsyncErrorSyndrome(errorstate : ErrorState) => bits(3)
begin
    case errorstate of
        when ErrorState\_UC => return '000';
        when ErrorState\_UEU => return '001';
        when ErrorState\_UEO => return '010';
        when ErrorState\_UER => return '011';
        when ErrorState\_CE => return '110';
        otherwise => unreachable;
    end;
end;
```

Library pseudocode for aarch64/functions/ras/AArch64_EncodeSyncErrorSyndrome

```
// AArch64_EncodeSyncErrorSyndrome()
// =====
// Return the encoding for specified ErrorState for a synchronous Abort
// exception taken to AArch64 state.

func AArch64_EncodeSyncErrorSyndrome(errorstate : ErrorState) => bits(2)
begin
    case errorstate of
        when ErrorState\_UC => return '10';
        when ErrorState\_UEU => return '10'; // UEU is reported as UC
        when ErrorState\_UEO => return '11';
        when ErrorState\_UER => return '00';
        otherwise => unreachable;
    end;
end;
```

Library pseudocode for aarch64/functions/ras/AArch64_PhysicalErrorSyndrome

```
// AArch64_PhysicalErrorSyndrome()
// =====
// Generate SError syndrome.

func AArch64_PhysicalErrorSyndrome(is_esb : boolean, implicit_esb : boolean) => bits(25)
begin
    var syndrome : bits(25) = Zeros{};
    var target_el : bits(2);
    (-, target_el) = PhysicalErrorTarget();

    if ReportErrorAsUncategorized() then
        syndrome = Zeros{25};
    elseif ReportErrorAsIMPDEF() then
        syndrome[24] = '1'; // IDS
        syndrome[23:0] = ImpDefBits{24}("IMPDEF ErrorState");
    else
        let fault : FaultRecord = GetPendingPhysicalError();
        let errorstate : ErrorState = PEErrorState(fault);
        syndrome[24] = '0'; // IDS
        if (!is_esb && IsFeatureImplemented(FEAT_PFAR) &&
            !(EL2Enabled() &&
              (HCR_EL2().VM == '1' || HCR_EL2().DC == '1') && target_el == EL1)) then
            syndrome[14] = if IsPFARValid(fault) then '1' else '0';
        end;
        syndrome[13] = if implicit_esb then '1' else '0'; // IESB
        syndrome[12:10] = AArch64\_EncodeAsyncErrorSyndrome(errorstate); // AET
        syndrome[9] = fault.extflag; // EA
        syndrome[5:0] = '010001'; // DFSC
    end;

    return syndrome;
end;
```

Library pseudocode for aarch64/functions/ras/AArch64_dESBOperation

```
// AArch64_dESBOperation()
// =====
// Perform the AArch64 ESB operation for a pending delegated SError exception.

func AArch64_dESBOperation()
begin
    assert (IsFeatureImplemented(FEAT_E3DSE) && !ELUsingAArch32(EL3) && PSTATE.EL != EL3);
    // When FEAT_E3DSE is implemented, SCR_EL3.DSE might inject a delegated SError exception.
    var dsei_pending, dsei_masked : boolean;
    dsei_pending = EffectiveSCR\_EL3\_EnDSE() == '1' && SCR_EL3().DSE == '1';
    (dsei_masked, -) = AArch64\_DelegatedSErrorTarget();
    if dsei_pending && dsei_masked then
        var target : bits(64) = Zeros{};
        target[31] = '1'; // A
        target[24:0] = VESR_EL3()[24:0];
        VDISR_EL3() = target;
        ClearPendingDelegatedSError();
    end;
    return;
end;
```

Library pseudocode for aarch64/functions/ras/AArch64_vESBOperation

```
// AArch64_vESBOperation()
// =====
// Perform the AArch64 ESB operation for an unmasked pending virtual SError exception.
// If FEAT_E3DSE is implemented and there is no unmasked virtual SError exception
// pending, then AArch64_dESBOperation() is called to perform the AArch64 ESB operation
// for a pending delegated SError exception.

func AArch64_vESBOperation()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2);

    // If physical SError exceptions are routed to EL2, and TGE is not set, then a virtual
    // SError exception might be pending.
    let vsei_pending : boolean = (IsVirtualSErrorPending() && EffectiveTGE() == '0' &&
        (EffectiveHCR\_AMO() == '1' || EffectiveHCRX\_EL2\_TMEA() == '1'));
    let vsei_masked : boolean = PSTATE.A == '1' || Halted() || ExternalDebugInterruptsDisabled(EL1);

    // Check for a masked virtual SError pending
    if vsei_pending && vsei_masked then
        // This function might be called for the interprocessing case, and INTdis is masking
        // the virtual SError exception.
        if ELUsingAArch32(EL1) then
            var target : bits(32) = Zeros{};
            target[31] = '1'; // A
            target[15:14] = VDFSR() [15:14]; // AET
            target[12] = VDFSR() [12]; // ExT
            target[9] = TTBCR().EAE; // LPAE
            if TTBCR().EAE == '1' then // Long-descriptor format
                target[5:0] = '010001'; // STATUS
            else // Short-descriptor format
                target[10,3:0] = '10110'; // FS
            end;
            VDISR() = target;
        else
            var target : bits(64) = Zeros{};
            target[31] = '1'; // A
            target[24:0] = VSESR\_EL2() [24:0];
            VDISR\_EL2() = target;
        end;
        ClearPendingVirtualSError();
    elseif IsFeatureImplemented(FEAT_E3DSE) then
        AArch64\_dESBOperation();
    end;

    return;
end;
```

Library pseudocode for aarch64/functions/ras/EffectiveSCR_EL3_EndSE

```
// EffectiveSCR_EL3_EndSE()
// =====
// Return the Effective value of SCR_EL3().EndSE.

func EffectiveSCR_EL3_EndSE() => bit
begin
    if !IsFeatureImplemented(FEAT_E3DSE) then
        return '0';
    end;
    if SCR\_EL3().EA == '0' && EffectiveSCR\_EL3\_TMEA() == '0' then
        return '0';
    end;
    return SCR\_EL3().EndSE;
end;
```

Library pseudocode for aarch64/functions/ras/FirstRecordOfNode

```
// FirstRecordOfNode()
// =====
// Return the first record in the node that contains the record n.

func FirstRecordOfNode(n : integer) => integer
begin
    for q = n downto 0 do
        if IsFirstRecordOfNode(q) then return q; end;
    end;
    unreachable;
end;
```

Library pseudocode for aarch64/functions/ras/IsCommonFaultInjectionImplemented

```
// IsCommonFaultInjectionImplemented()
// =====
// Check if the Common Fault Injection Model Extension is implemented by the node that owns this
// error record.

impdef func IsCommonFaultInjectionImplemented(n : integer) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for aarch64/functions/ras/IsCountableErrorsRecorded

```
// IsCountableErrorsRecorded()
// =====
// Check whether Error record n records countable errors.

impdef func IsCountableErrorsRecorded(n : integer) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for aarch64/functions/ras/IsErrorAddressIncluded

```
// IsErrorAddressIncluded()
// =====
// Check whether Error record n includes an address associated with an error.

impdef func IsErrorAddressIncluded(n : integer) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for aarch64/functions/ras/IsErrorRecordImplemented

```
// IsErrorRecordImplemented()
// =====
// Is the error record n implemented

impdef func IsErrorRecordImplemented(n : integer) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for aarch64/functions/ras/IsFirstRecordOfNode

```
// IsFirstRecordOfNode()
// =====
// Check if the record q is the first error record in its node.

impdef func IsFirstRecordOfNode(q : integer) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for aarch64/functions/ras/IsPFARValid

```
// IsPFARValid()
// =====
// Determine if faulting PA can be reported.

impdef func IsPFARValid(fault : FaultRecord) => boolean
begin
    return !IsAsyncAbort(fault);
end;
```

Library pseudocode for aarch64/functions/ras/IsSPMUCounterImplemented

```
// IsSPMUCounterImplemented()
// =====
// Does the System PMU s implement the counter n.

impdef func IsSPMUCounterImplemented(s : integer, n : integer) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for aarch64/functions/rcw

```
constant RCW64_PROTECTED_BIT : integer = 52;
constant RCW128_PROTECTED_BIT : integer = 114;
```

Library pseudocode for aarch64/functions/rcw/ProtectionEnabled

```
// ProtectionEnabled()
// =====
// Returns TRUE if the ProtectedBit is
// enabled in the current Exception level.

func ProtectionEnabled(el : bits(2)) => boolean
begin
    assert HaveEL(el);
    let regime : bits(2) = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));
    if (!IsD128Enabled(el)) then
        case regime of
            when EL1 =>
                return IsTCR2EL1Enabled() && TCR2_EL1().PnCH == '1';
            when EL2 =>
                return IsTCR2EL2Enabled() && TCR2_EL2().PnCH == '1';
            when EL3 =>
                return TCR_EL3().PnCH == '1';
        end;
    else
        return TRUE;
    end;
    return FALSE;
end;
```

Library pseudocode for aarch64/functions/rcw/RCWCheck

```
// RCWCheck()
// =====
// Returns nzcvc based on : if the new value for RCW/RCWS instructions satisfy RCW and/or RCWS checks
// Z is set to 1 if RCW checks fail
// C is set to 0 if RCWS checks fail

func RCWCheck{N}(old : bits(N), new : bits(N), soft : boolean) => bits(4)
begin
    assert N IN {64,128};
    let protectedbit : integer = if N == 128 then RCW128\_PROTECTED\_BIT else RCW64\_PROTECTED\_BIT;
    var rcw_fail : boolean = FALSE;
    var rcws_fail : boolean = FALSE;
    var rcw_state_fail : boolean = FALSE;
    var rcws_state_fail : boolean = FALSE;
    var rcw_mask_fail : boolean = FALSE;
    var rcws_mask_fail : boolean = FALSE;

    //Effective RCWMask calculation
    var rcwmask : bits(N) = RCWMASK_EL1() [N-1:0];
    if N == 64 then
        rcwmask[49:18] = Replicate{32}(rcwmask[17]);
        rcwmask[0] = '0';
    else
        rcwmask[55:17] = Replicate{39}(rcwmask[16]);
        rcwmask[126:125,120:119,107:101,90:56,1:0] = Zeros{48};
    end;

    //Effective RCWSMask calculation
    var rcwsoftmask : bits(N) = RCWSMASK_EL1() [N-1:0];
    if N == 64 then
        rcwsoftmask[49:18] = Replicate{32}(rcwsoftmask[17]);
        rcwsoftmask[0] = '0';
        if (ProtectionEnabled(PSTATE.EL)) then
            rcwsoftmask[52] = '0';
        end;
    else
        rcwsoftmask[55:17] = Replicate{39}(rcwsoftmask[16]);
        rcwsoftmask[126:125,120:119,107:101,90:56,1:0] = Zeros{48};
        rcwsoftmask[114] = '0';
    end;

    //RCW Checks
    //State Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old[protectedbit] == '1' then
            rcw_state_fail = new[protectedbit,0] != old[protectedbit,0];
        elseif old[protectedbit] == '0' then
            rcw_state_fail = new[protectedbit] != old[protectedbit];
        end;
    end;

    //Mask Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old[protectedbit,0] == '11' then
            rcw_mask_fail = !IsZero((new XOR old) AND NOT(rcwmask));
        end;
    end;

    //RCWS Checks
    if soft then
        //State Check
        if old[0] == '1' then
            rcws_state_fail = new[0] != old[0];
        elseif (!ProtectionEnabled(PSTATE.EL) ||
            (ProtectionEnabled(PSTATE.EL) && old[protectedbit] == '0')) then
            rcws_state_fail = new[0] != old[0];
        end;
        //Mask Check
        if old[0] == '1' then
            rcws_mask_fail = !IsZero((new XOR old) AND NOT(rcwsoftmask));
        end;
    end;

    rcw_fail = rcw_state_fail || rcw_mask_fail;
    rcws_fail = rcws_state_fail || rcws_mask_fail;

    let n : bit = '0';
    let z : bit = if rcw_fail then '1' else '0';
    let c : bit = if rcws_fail then '0' else '1';
    let v : bit = '0';
    return n::z::c::v;
end;
```

Library pseudocode for aarch64/functions/reduceop/FPReduce

```
// FPReduce()
// =====
// Perform the floating-point operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result.

func FPReduce(esize, N)(op : ReduceOp, input : bits(N),
                        fpcr : FPCR_Type) => bits(esize) recurselimit 8
begin
    var hi : bits(esize);
    var lo : bits(esize);
    var result : bits(esize);
    let half : integer{} = N DIV 2;

    if N == esize then
        return input[esize-1:0];
    end;

    hi = FPReduce{esize, (N - N DIV 2)}(op, input[N-1:half], fpcr);
    lo = FPReduce{esize, N DIV 2}(op, input[half-1:0], fpcr);
    case op of
        when ReduceOp\_FMINNUM =>
            result = FPMinNum{esize}(lo, hi, fpcr);
        when ReduceOp\_FMAXNUM =>
            result = FPMaxNum{esize}(lo, hi, fpcr);
        when ReduceOp\_FMIN =>
            result = FPMin{esize}(lo, hi, fpcr);
        when ReduceOp\_FMAX =>
            result = FPMax{esize}(lo, hi, fpcr);
        when ReduceOp\_FADD =>
            result = FPAdd{esize}(lo, hi, fpcr);
    end;

    return result;
end;
```

Library pseudocode for aarch64/functions/reduceop/IntReduce

```
// IntReduce()
// =====
// Perform the integer operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result.

func IntReduce(esize, N)(op : ReduceOp, input : bits(N)) => bits(esize) recurselimit 8
begin
    var hi : bits(esize);
    var lo : bits(esize);
    var result : bits(esize);
    let half : integer{} = N DIV 2;
    if N == esize then
        return input[esize-1:0];
    end;

    hi = IntReduce{esize, (N - N DIV 2)}(op, input[N-1:half]);
    lo = IntReduce{esize, N DIV 2}(op, input[half-1:0]);
    case op of
        when ReduceOp\_ADD =>
            result = lo + hi;
    end;

    return result;
end;
```

Library pseudocode for aarch64/functions/reduceop/ReduceOp

```
// ReduceOp
// =====
// Vector reduce instruction types.

type ReduceOp of enumeration {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                             ReduceOp_FMIN, ReduceOp_FMAX,
                             ReduceOp_FADD, ReduceOp_ADD};
```

Library pseudocode for aarch64/functions/registers/AArch64_MaybeZeroRegisterUppers

```
// AArch64_MaybeZeroRegisterUppers()
// =====
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

func AArch64_MaybeZeroRegisterUppers()
begin
    assert UsingAArch32(); // Always called from AArch32 state before entering AArch64 state

    var first : integer;
    var last : integer;
    var include_R15 : boolean;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0; last = 14; include_R15 = FALSE;
    elseif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0; last = 30; include_R15 = FALSE;
    else
        first = 0; last = 30; include_R15 = TRUE;
    end;

    for n = first to last do
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            R[n][63:32] = Zeros{32};
        end;
    end;

    return;
end;
```

Library pseudocode for aarch64/functions/registers/AArch64_ResetGeneralRegisters

```
// AArch64_ResetGeneralRegisters()
// =====

func AArch64_ResetGeneralRegisters()
begin
    for i = 0 to 30 do
        X{64}(i) = ARBITRARY : bits(64);
    end;
end;
```

Library pseudocode for aarch64/functions/registers/AArch64_ResetSIMDFPRegisters

```
// AArch64_ResetSIMDFPRegisters()
// =====

func AArch64_ResetSIMDFPRegisters()
begin
    for i = 0 to 31 do
        V{128}(i) = ARBITRARY : bits(128);
    end;
end;
```


Library pseudocode for aarch64/functions/registers/AArch64_ResetSpecialRegisters

```
// AArch64_ResetSpecialRegisters()
// =====

func AArch64_ResetSpecialRegisters()
begin

    // AArch64 special registers
    SP_EL0() = ARBITRARY : bits(64);
    SP_EL1() = ARBITRARY : bits(64);

    SPSR_EL1() = ARBITRARY : bits(64);
    ELR_EL1() = ARBITRARY : bits(64);
    if HaveEL(EL2) then
        SP_EL2() = ARBITRARY : bits(64);
        SPSR_EL2() = ARBITRARY : bits(64);
        ELR_EL2() = ARBITRARY : bits(64);
    end;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq()[31:0] = ARBITRARY : bits(32);
        SPSR_irq()[31:0] = ARBITRARY : bits(32);
        SPSR_abt()[31:0] = ARBITRARY : bits(32);
        SPSR_und()[31:0] = ARBITRARY : bits(32);
    end;

    // External debug special registers
    DLR_EL0() = ARBITRARY : bits(64);
    DSPSR_EL0() = ARBITRARY : bits(64);

    return;
end;
```

Library pseudocode for aarch64/functions/registers/AArch64_ResetSystemRegisters

```
// AArch64_ResetSystemRegisters()
// =====

impdef func AArch64_ResetSystemRegisters(cold_reset : boolean)
begin
    return;
end;
```

Library pseudocode for aarch64/functions/registers/PC64

```
// PC64 - getter
// =====
// Read 64-bit program counter.

func PC64() => bits(64)
begin
    return PC;
end;
```

Library pseudocode for aarch64/functions/registers/SP

```
// SP - accessor
// =====

accessor SP{width}() <=> value : bits(width)
begin
    // Write a 32-bit or 64-bit value to the current stack pointer.
    setter
        assert width IN {64, 32};
        if PSTATE.SP == '0' then
            SP_EL0() = ZeroExtend{64}(value);
        else
            case PSTATE.EL of
                when EL0 => SP_EL0() = ZeroExtend{64}(value);
                when EL1 => SP_EL1() = ZeroExtend{64}(value);
                when EL2 => SP_EL2() = ZeroExtend{64}(value);
                when EL3 => SP_EL3() = ZeroExtend{64}(value);
            end;
        end;
    end;

    // Read the least-significant 32 or 64 bits from the current stack pointer.
    getter
        assert width IN {64, 32};
        if PSTATE.SP == '0' then
            return SP_EL0()[width-1:0];
        else
            case PSTATE.EL of
                when EL0 => return SP_EL0()[width-1:0];
                when EL1 => return SP_EL1()[width-1:0];
                when EL2 => return SP_EL2()[width-1:0];
                when EL3 => return SP_EL3()[width-1:0];
            end;
        end;
    end;
end;
```

Library pseudocode for aarch64/functions/registers/X

```
// X - accessor
// =====

accessor X{width}(n : integer) <=> value : bits(width)
begin
    // Write a 32-bit or 64-bit value to a general-purpose register.
    setter
        assert n >= 0 && n <= 31;
        assert width IN {32, 64};
        if n != 31 then
            _R[[n]] = ZeroExtend{64}(value);
        end;
        return;
    end;

    // Read the least-significant 8, 16, 32, or 64 bits from a general-purpose register.
    getter
        assert n >= 0 && n <= 31;
        let rw : integer{} = width as integer{8, 16, 32, 64};
        if n != 31 then
            return _R[[n]][rw-1:0];
        else
            return Zeros{rw};
        end;
    end;
end;

// X - accessor
// =====

accessor X{width}(lr : integer, hr : integer) <=> value : bits(width)
begin
    // Write a 128-bit value to two general-purpose registers.
    setter
        assert width == 128;
        let half : integer{} = width DIV 2;
        X{half}(lr) = value[0+:half];
        X{half}(hr) = value[half+:half];
        return;
    end;

    // Read 128 bits from two separate general-purpose registers.
    getter
        assert width == 128;
        let half : integer{} = width DIV 2;
        return X{half}(hr) :: X{half}(lr);
    end;
end;
```

Library pseudocode for aarch64/functions/shiftreg/DecodeShift

```
// DecodeShift()
// =====
// Decode shift encodings

func DecodeShift(op : bits(2)) => ShiftType
begin
  case op of
    when '00' => return ShiftType_LSL;
    when '01' => return ShiftType_LSR;
    when '10' => return ShiftType_ASR;
    when '11' => return ShiftType_ROR;
  end;
end;
```

Library pseudocode for aarch64/functions/shiftreg/ShiftReg

```
// ShiftReg()
// =====
// Perform shift of a register operand

func ShiftReg{N}(reg : integer, shifttype : ShiftType, amount : integer) => bits(N)
begin
  var result : bits(N) = X{}(reg);
  case shifttype of
    when ShiftType_LSL => result = LSL(result, amount);
    when ShiftType_LSR => result = LSR(result, amount);
    when ShiftType_ASR => result = ASR(result, amount);
    when ShiftType_ROR => result = ROR(result, amount);
  end;
  return result;
end;
```

Library pseudocode for aarch64/functions/shiftreg/ShiftType

```
// ShiftType
// =====
// AArch64 register shifts.

type ShiftType of enumeration {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

Library pseudocode for aarch64/functions/sme/CounterToPredicate

```
// CounterToPredicate()
// =====

func CounterToPredicate{width}(pred : bits(16)) => bits(width)
begin
  var count : integer;
  var esize : ESize;
  var elements : integer;
  let VL : VecLen = CurrentVL();
  let PL : PredLen = VL DIV 8;
  let maxbit : integer{} = FloorLog2(PL * 4) as integer{0..14};
  var result : bits(PL*4);
  let invert : boolean = pred[15] == '1';

  assert width == PL || width == PL*2 || width == PL*3 || width == PL*4;

  case pred[3:0] of
    when '0000' =>
      return Zeros{width};
    when 'xxx1' =>
      count = UInt(pred[maxbit:1]);
      esize = 8;
    when 'xx10' =>
      count = UInt(pred[maxbit:2]);
      esize = 16;
    when 'x100' =>
      count = UInt(pred[maxbit:3]);
      esize = 32;
    when '1000' =>
      count = UInt(pred[maxbit:4]);
      esize = 64;
  end;

  elements = (VL * 4) DIV esize;
  result = Zeros{PL*4};
  let psize : integer{} = esize DIV 8;
  for e = 0 to elements-1 do
    var pbit : bit = if e < count then '1' else '0';
    if invert then
      pbit = NOT(pbit);
    end;
    result[e*psize] = ZeroExtend{psize}(pbit);
  end;

  return result[width-1:0];
end;
```

Library pseudocode for aarch64/functions/sme/EncodePredCount

```
// EncodePredCount()
// =====

func EncodePredCount{width}(esize : ESize, elements : integer,
                             count_in : integer, invert_in : boolean) => bits(width)
begin
  var count : integer = count_in;
  var invert : boolean = invert_in;
  let PL : PredLen = CurrentVL() DIV 8;
  assert width == PL;
  assert count >= 0 && count <= elements;
  var pred : bits(16);

  if count == 0 then
    return Zeros{width};
  end;

  if invert then
    count = elements - count;
  elseif count == elements then
    count = 0;
    invert = TRUE;
  end;

  let inv : bit = (if invert then '1' else '0');
  case esize of
    when 8 => pred = inv :: count[13:0] :: '1';
    when 16 => pred = inv :: count[12:0] :: '10';
    when 32 => pred = inv :: count[11:0] :: '100';
    when 64 => pred = inv :: count[10:0] :: '1000';
  end;

  return ZeroExtend{width}(pred);
end;
```

Library pseudocode for aarch64/functions/sme/Lookup

```
// Lookup Table
// =====

var _ZT0 : bits(ZT0\_LEN);
```

Library pseudocode for aarch64/functions/sme/PredCountTest

```
// PredCountTest()
// =====

func PredCountTest(elements : integer, count : integer, invert : boolean) => bits(4)
begin
  var n, z, c, v : bit;
  z = (if count == 0 then '1' else '0');           // none active
  if !invert then
    n = (if count != 0 then '1' else '0');         // first active
    c = (if count == elements then '0' else '1');  // NOT last active
  else
    n = (if count == elements then '1' else '0');  // first active
    c = (if count != 0 then '0' else '1');         // NOT last active
  end;
  v = '0';

  return n::z::c::v;
end;
```

Library pseudocode for aarch64/functions/sme/System

```
// System Registers
// =====

var _ZA : array [[256]] of bits(MAX\_VL);
```

Library pseudocode for aarch64/functions/sme/ZAhslice

```
// ZAhslice - accessor
// =====

accessor ZAhslice{width}(tile : integer, esize : ESize, slice : integer) <=> value : bits(width)
begin
  getter
    let tiles : integer = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    let slices : integer = CurrentSVL() DIV esize;
    assert slice >= 0 && slice < slices;

    return ZAvector{width}(tile + slice * tiles);
  end;
  setter
    let tiles : integer = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    let slices : integer = CurrentSVL() DIV esize;
    assert slice >= 0 && slice < slices;

    ZAvector{width}(tile + slice * tiles) = value;
  end;
end;
```

Library pseudocode for aarch64/functions/sme/ZAslice

```
// ZAslice - accessor
// =====

accessor ZAslice{width}(tile : integer, esize : ESize,
                        vertical : boolean, slice : integer) <=> value : bits(width)
begin
  getter
    var result : bits(width);

    if vertical then
      result = ZAvslice{width}(tile, esize, slice);
    else
      result = ZAhslice{width}(tile, esize, slice);
    end;

    return result;
  end;
  setter
    if vertical then
      ZAvslice{width}(tile, esize, slice) = value;
    else
      ZAhslice{width}(tile, esize, slice) = value;
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/sme/ZAtile

```
// ZAtile - accessor
// =====

accessor ZAtile{width}(tile : integer, esize : ESize) <=> value : bits(width)
begin
  getter
    let SVL : VecLen = CurrentSVL();
    let slices : integer = SVL DIV esize;
    assert width == SVL * slices;
    var result : bits(width);

    for slice = 0 to slices-1 do
      result[slice*SVL] = ZAhslice{SVL}(tile, esize, slice);
    end;

    return result;
  end;
  setter
    let SVL : VecLen = CurrentSVL();
    let slices : integer = SVL DIV esize;
    assert width == SVL * slices;

    for slice = 0 to slices-1 do
      ZAhslice{SVL}(tile, esize, slice) = value[slice*SVL];
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/sme/ZAvector

```
// ZAvector - accessor
// =====

accessor ZAvector{width}(index : integer) <=> value : bits(width)
begin
  getter
    assert width == CurrentSVL();
    assert index >= 0 && index < (width DIV 8);

    return ZA[[index]][width-1:0];
  end;
  setter
    assert width == CurrentSVL();
    assert index >= 0 && index < (width DIV 8);

    if ConstrainUnpredictableBool(Unpredictable SMEZEROUPPER) then
      ZA[[index]] = ZeroExtend{MAX VL}(value);
    else
      ZA[[index]][width-1:0] = value;
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/sme/ZAvslice

```
// ZAvslice - accessor
// =====

accessor ZAvslice{width}(tile : integer, esize : ESize, slice : integer) <=> value : bits(width)
begin
  getter
    let slices : integer = CurrentSVL() DIV esize;
    var result : bits(width);

    for s = 0 to slices-1 do
      let hslice : bits(width) = ZAhslice{}(tile, esize, s);
      result[s*:esize] = hslice[slice*:esize];
    end;

    return result;
  end;

  setter
    let slices : integer = CurrentSVL() DIV esize;

    for s = 0 to slices-1 do
      var hslice : bits(width) = ZAhslice{}(tile, esize, s);
      hslice[slice*:esize] = value[s*:esize];
      ZAhslice{width}(tile, esize, s) = hslice;
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/sme/ZT0

```
// ZT0 - accessor
// =====

accessor ZT0{width}() <=> value : bits(width)
begin
  getter
    assert width == ZT0\_LEN;
    return ZT0[width-1:0];
  end;
  setter
    assert width == ZT0\_LEN;
    ZT0[width-1:0] = value;
  end;
end;
```

Library pseudocode for aarch64/functions/sve/AArch32_IsFPEEnabled

```
// AArch32_IsFPEEnabled()
// =====
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

func AArch32_IsFPEEnabled(el : bits(2)) => boolean
begin
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64\_IsFPEEnabled(el);
    end;

    if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS\_NonSecure then
        // Check if access disabled in NSACR
        if NSACR().cp10 == '0' then return FALSE; end;
    end;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        var disabled : boolean;
        case CPACR().cp10 of
            when '00' => disabled = TRUE;
            when '01' => disabled = el == EL0;
            when '10' => disabled = ConstrainUnpredictableBool(Unpredictable\_RESCPACR);
            when '11' => disabled = FALSE;
        end;
        if disabled then return FALSE; end;
    end;

    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            return AArch64\_IsFPEEnabled(EL2);
        end;
        if HCPTR().TCP10 == '1' then return FALSE; end;
    end;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR\_EL3().TFP == '1' then return FALSE; end;
    end;

    return TRUE;
end;
```

Library pseudocode for aarch64/functions/sve/AArch64_IsFPEEnabled

```
// AArch64_IsFPEEnabled()
// =====
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

func AArch64_IsFPEEnabled(el : bits(2)) => boolean
begin
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SIMD&FP at EL0/EL1
        var disabled : boolean;
        case CPACR\_EL1().FPEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = el == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then return FALSE; end;
    end;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            var disabled : boolean;
            case CPTR\_EL2().FPEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = el == EL0 && HCR\_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then return FALSE; end;
        else
            if CPTR\_EL2().TFP == '1' then return FALSE; end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR\_EL3().TFP == '1' then return FALSE; end;
    end;

    return TRUE;
end;
```


Library pseudocode for aarch64/functions/sve/ActivePredicateElement

```
// ActivePredicateElement()
// =====
// Returns TRUE if the predicate bit is 1 and FALSE otherwise

func ActivePredicateElement{N}(pred : bits(N), e : integer, esize : integer) => boolean
begin
    assert esize IN {8, 16, 32, 64, 128};
    let n : integer = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred[n] == '1';
end;
```

Library pseudocode for aarch64/functions/sve/AllElementsActive

```
// AllElementsActive()
// =====
// Return TRUE if all the elements are active in the mask. Otherwise,
// return FALSE.

func AllElementsActive{N}(mask : bits(N), esize : integer) => boolean
begin
    let elements : integer = N DIV (esize DIV 8);
    var active : integer = 0;
    for e = 0 to elements-1 do
        if ActivePredicateElement{N}(mask, e, esize) then active = active + 1; end;
    end;
    return active == elements;
end;
```

Library pseudocode for aarch64/functions/sve/AnyActiveElement

```
// AnyActiveElement()
// =====
// Return TRUE if there is at least one active element in mask. Otherwise,
// return FALSE.

func AnyActiveElement{N}(mask : bits(N), esize : integer) => boolean
begin
    return LastActiveElement{N}(mask, esize) >= 0;
end;
```

Library pseudocode for aarch64/functions/sve/BitDeposit

```
// BitDeposit()
// =====
// Deposit the least significant bits from DATA into result positions
// selected by nonzero bits in MASK, setting other result bits to zero.

func BitDeposit{N}(data : bits(N), mask : bits(N)) => bits(N)
begin
    var res : bits(N) = Zeros{};
    var db : integer = 0;
    for rb = 0 to N-1 do
        if mask[rb] == '1' then
            res[rb] = data[db];
            db = db + 1;
        end;
    end;
    return res;
end;
```

Library pseudocode for aarch64/functions/sve/BitExtract

```
// BitExtract()
// =====
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, setting other result bits to zero.

func BitExtract{N}(data : bits(N), mask : bits(N)) => bits(N)
begin
    var res : bits(N) = Zeros{};
    var rb : integer = 0;
    for db = 0 to N-1 do
        if mask[db] == '1' then
            res[rb] = data[db];
            rb = rb + 1;
        end;
    end;
    return res;
end;
```

Library pseudocode for aarch64/functions/sve/BitGroup

```
// BitGroup()
// =====
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, and pack unselected bits into the
// most significant result bits.

func BitGroup{N}(data : bits(N), mask : bits(N)) => bits(N)
begin
    var res : bits(N);
    var rb : integer = 0;

    // compress masked bits to right
    for db = 0 to N-1 do
        if mask[db] == '1' then
            res[rb] = data[db];
            rb      = rb + 1;
        end;
    end;
    // compress unmasked bits to left
    for db = 0 to N-1 do
        if mask[db] == '0' then
            res[rb] = data[db];
            rb      = rb + 1;
        end;
    end;
    return res;
end;
```

Library pseudocode for aarch64/functions/sve/CheckNonStreamingSVEEnabled

```
// CheckNonStreamingSVEEnabled()
// =====
// Checks for traps on SVE instructions that are not legal when executed in Streaming mode.

func CheckNonStreamingSVEEnabled()
begin
    CheckSVEEnabled\(\);

    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64Enabled\(\) then
        SMEAccessTrap\(SMEExceptionType Streaming, PSTATE.EL\);
    end;
end;
```

Library pseudocode for aarch64/functions/sve/CheckOriginalSVEEnabled

```
// CheckOriginalSVEEnabled()
// =====
// Checks for traps on SVE instructions and instructions that access SVE System
// registers.

func CheckOriginalSVEEnabled()
begin
    assert IsFeatureImplemented(FEAT_SVE);
    var disabled : boolean;

    if (HaveEL(EL3) && (CPTR_EL3().EZ == '0' || CPTR_EL3().TFP == '1') &&
        EL3SDDUndefPriority()) then
        Undefined();
    end;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1().ZEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then SVEAccessTrap(EL1); end;

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1().FPEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then AArch64_AdvSIMDFPAccessTrap(EL1); end;
    end;

    // Check if access disabled in CPTR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            // Check SVE at EL2
            case CPTR_EL2().ZEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then SVEAccessTrap(EL2); end;

            // Check SIMD&FP at EL2
            case CPTR_EL2().FPEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        else
            if CPTR_EL2().TZ == '1' then SVEAccessTrap(EL2); end;
            if CPTR_EL2().TFP == '1' then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().EZ == '0' then
            if EL3SDDUndef() then Undefined(); end;
            SVEAccessTrap(EL3);
        end;

        if CPTR_EL3().TFP == '1' then
            if EL3SDDUndef() then Undefined(); end;
            AArch64_AdvSIMDFPAccessTrap(EL3);
        end;
    end;
end;
```

Library pseudocode for aarch64/functions/sve/CheckSMEAccess

```
// CheckSMEAccess()
// =====
// Check that access to SME System registers is enabled.

func CheckSMEAccess()
begin
    var disabled : boolean;

    if HaveEL(EL3) && CPTR_EL3().ESM == '0' && EL3SDDUndefPriority() then
        Undefined();
    end;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1().SMEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then SMEAccessTrap(SMEExceptionType AccessTrap, EL1); end;
    end;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            // Check SME at EL2
            case CPTR_EL2().SMEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then SMEAccessTrap(SMEExceptionType AccessTrap, EL2); end;
        else
            if CPTR_EL2().TSM == '1' then SMEAccessTrap(SMEExceptionType AccessTrap, EL2); end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().ESM == '0' then
            if EL3SDDUndef() then Undefined(); end;
            SMEAccessTrap(SMEExceptionType AccessTrap, EL3);
        end;
    end;
end;
```

Library pseudocode for aarch64/functions/sve/CheckSMEAndZAAEnabled

```
// CheckSMEAndZAAEnabled()
// =====

func CheckSMEAndZAAEnabled()
begin
    CheckSMEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType InactiveZA, PSTATE.EL);
    end;
end;
```

Library pseudocode for aarch64/functions/sve/CheckSMEEnabled

```
// CheckSMEEnabled()
// =====

func CheckSMEEnabled()
begin
    var disabled : boolean;

    if HaveEL(EL3) && CPTR_EL3().[ESM,TFP] != '10' && EL3SDDUndefPriority() then
        Undefined();
    end;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1().SMEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then SMEAccessTrap(SMEExceptionType AccessTrap, EL1); end;

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1().FPEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then AArch64_AdvSIMDFPAccessTrap(EL1); end;
    end;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            // Check SME at EL2
            case CPTR_EL2().SMEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then SMEAccessTrap(SMEExceptionType AccessTrap, EL2); end;

            // Check SIMD&FP at EL2
            case CPTR_EL2().FPEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = PSTATE.EL == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        else
            if CPTR_EL2().TSM == '1' then SMEAccessTrap(SMEExceptionType AccessTrap, EL2); end;
            if CPTR_EL2().TFP == '1' then AArch64_AdvSIMDFPAccessTrap(EL2); end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().ESM == '0' then
            if EL3SDDUndef() then Undefined(); end;
            SMEAccessTrap(SMEExceptionType AccessTrap, EL3);
        end;

        if CPTR_EL3().TFP == '1' then
            if EL3SDDUndef() then Undefined(); end;
            AArch64_AdvSIMDFPAccessTrap(EL3);
        end;
    end;
end;
```

Library pseudocode for aarch64/functions/sve/CheckSMEZT0Enabled

```
// CheckSMEZT0Enabled()
// =====
// Checks for ZT0 enabled.

func CheckSMEZT0Enabled()
begin
    if HaveEL(EL3) && SMCR_EL3().EZT0 == '0' && EL3SDDUndefPriority() then
        Undefined();
    end;

    // Check if ZA and ZT0 are inactive in PSTATE
    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
    end;

    // Check if EL0/EL1 accesses to ZT0 are disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        if SMCR_EL1().EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL1);
        end;
    end;

    // Check if EL0/EL1/EL2 accesses to ZT0 are disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2().EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL2);
        end;
    end;

    // Check if all accesses to ZT0 are disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3().EZT0 == '0' then
            if EL3SDDUndef() then Undefined(); end;
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL3);
        end;
    end;
end;
```

Library pseudocode for aarch64/functions/sve/CheckSVEEnabled

```
// CheckSVEEnabled()
// =====
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

func CheckSVEEnabled()
begin
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        CheckSMEEnabled();
    elsif IsFeatureImplemented(FEAT_SME) && !IsFeatureImplemented(FEAT_SVE) then
        CheckStreamingSVEEnabled();
    else
        CheckOriginalSVEEnabled();
    end;
end;
```

Library pseudocode for aarch64/functions/sve/CheckStreamingSVEAndZAEEnabled

```
// CheckStreamingSVEAndZAEEnabled()
// =====

func CheckStreamingSVEAndZAEEnabled()
begin
    CheckStreamingSVEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
    end;
end;
```

Library pseudocode for aarch64/functions/sve/CheckStreamingSVEEnabled

```
// CheckStreamingSVEEnabled()
// =====

func CheckStreamingSVEEnabled()
begin
    CheckSMEEnabled();

    if PSTATE.SM == '0' then
        SMEAccessTrap(SMEExceptionType_NotStreaming, PSTATE.EL);
    end;
end;
```

Library pseudocode for aarch64/functions/sve/CmpOp

```
// CmpOp
// =====

type CmpOp of enumeration { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

Library pseudocode for aarch64/functions/sve/CurrentNSVL

```
// CurrentNSVL
// =====
// Non-Streaming VL

readonly func CurrentNSVL() => VecLen
begin
    var vl : integer;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(ZCR\_EL1() .LEN);
    end;

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(ZCR\_EL2() .LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(ZCR\_EL2() .LEN));
    end;

    if PSTATE.EL == EL3 then
        vl = UInt(ZCR\_EL3() .LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR\_EL3() .LEN));
    end;

    return ImplementedSVEVectorLength((vl + 1) * 128);
end;
```

Library pseudocode for aarch64/functions/sve/CurrentSVL

```
// CurrentSVL
// =====
// Streaming SVL

readonly func CurrentSVL() => VecLen
begin
    var vl : integer;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(SMCR\_EL1() .LEN);
    end;

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(SMCR\_EL2() .LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(SMCR\_EL2() .LEN));
    end;

    if PSTATE.EL == EL3 then
        vl = UInt(SMCR\_EL3() .LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(SMCR\_EL3() .LEN));
    end;

    return ImplementedSMEVectorLength((vl + 1) * 128);
end;
```

Library pseudocode for aarch64/functions/sve/CurrentVL

```
// CurrentVL
// =====

readonly func CurrentVL() => VecLen
begin
    return (if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then CurrentSVL()
        else CurrentNSVL() );
end;
```

Library pseudocode for aarch64/functions/sve/DecodePredCount

```
// DecodePredCount()
// =====

func DecodePredCount(bitpattern : bits(5), esize : integer) => integer
begin
    let elements : integer = CurrentVL() DIV esize;
    var numElem : integer;
    case bitpattern of
        when '00000' => numElem = FloorPow2(elements);
        when '00001' => numElem = if elements >= 1 then 1 else 0;
        when '00010' => numElem = if elements >= 2 then 2 else 0;
        when '00011' => numElem = if elements >= 3 then 3 else 0;
        when '00100' => numElem = if elements >= 4 then 4 else 0;
        when '00101' => numElem = if elements >= 5 then 5 else 0;
        when '00110' => numElem = if elements >= 6 then 6 else 0;
        when '00111' => numElem = if elements >= 7 then 7 else 0;
        when '01000' => numElem = if elements >= 8 then 8 else 0;
        when '01001' => numElem = if elements >= 16 then 16 else 0;
        when '01010' => numElem = if elements >= 32 then 32 else 0;
        when '01011' => numElem = if elements >= 64 then 64 else 0;
        when '01100' => numElem = if elements >= 128 then 128 else 0;
        when '01101' => numElem = if elements >= 256 then 256 else 0;
        when '11101' => numElem = elements - (elements MOD 4);
        when '11110' => numElem = elements - (elements MOD 3);
        when '11111' => numElem = elements;
        otherwise => numElem = 0;
    end;
    return numElem;
end;
```

Library pseudocode for aarch64/functions/sve/ElemFFR

```
// ElemFFR - accessor
// =====

accessor ElemFFR(e : integer, esize : ESize) <=> value : bit
begin
    getter
        return PredicateElement{MAX\_PL}( FFR, e, esize);
    end;
    setter
        let psize : integer{} = esize DIV 8;
        FFR[e*:psize] = ZeroExtend{psize}(value);
    end;
end;
```

Library pseudocode for aarch64/functions/sve/FFR

```
// FFR - accessor
// =====

accessor FFR(width){} <=> value : bits(width)
begin
    getter
        assert width == CurrentVL() DIV 8;
        return FFR[width-1:0];
    end;

    setter
        assert width == CurrentVL() DIV 8;
        if ConstrainUnpredictableBool(Unpredictable SVEZEROUPPER) then
            FFR = ZeroExtend{MAX\_PL}(value);
        else
            FFR[width-1:0] = value;
        end;
    end;
end;
```


Library pseudocode for aarch64/functions/sve/FPCCompareNE

```
// FPCCompareNE()
// =====

func FPCCompareNE{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {16,32,64};
    var result : boolean;
    let (type1,sign1,value1) : (FPTType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPTType, bit, real) = FPUnpack{N}(op2, fpcr);
    let op1_nan : boolean = type1 IN {FPTType_SNaN, FPTType_QNaN};
    let op2_nan : boolean = type2 IN {FPTType_SNaN, FPTType_QNaN};

    if op1_nan || op2_nan then
        result = TRUE;
        if type1 == FPTType_SNaN || type2 == FPTType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
        end;
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
        FPProcessDenorms(type1, type2, N, fpcr);
    end;
    return result;
end;
```

Library pseudocode for aarch64/functions/sve/FPCCompareUN

```
// FPCCompareUN()
// =====

func FPCCompareUN{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {16,32,64};
    let (type1,sign1,value1) : (FPTType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPTType, bit, real) = FPUnpack{N}(op2, fpcr);

    if type1 == FPTType_SNaN || type2 == FPTType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
    end;

    let result : boolean = (type1 IN {FPTType_SNaN, FPTType_QNaN} ||
                           type2 IN {FPTType_SNaN, FPTType_QNaN});
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);
    end;

    return result;
end;
```

Library pseudocode for aarch64/functions/sve/FPCConvertSVE

```
// FPCConvertSVE()
// =====

func FPCConvertSVE{M, N}(op : bits(N), fpcr_in : FPCR_Type, rounding : FPRounding) => bits(M)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    return FPCConvert{M, N}(op, fpcr, rounding);
end;

// FPCConvertSVE()
// =====

func FPCConvertSVE{M, N}(op : bits(N), fpcr_in : FPCR_Type) => bits(M)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    return FPCConvert{M, N}(op, fpcr, FPRoundingMode(fpcr));
end;
```

Library pseudocode for aarch64/functions/sve/FPExpA

```
// FPExpA()
// =====

func FPExpA{N}(op : bits(N)) => bits(N)
begin
    assert N IN {16,32,64};
    var result : bits(N);
    var coeff : bits(N);
    let idx : integer = if N == 16 then UInt(op[4:0]) else UInt(op[5:0]);
    coeff = FPExpCoefficient{N}(idx);
    if N == 16 then
        result[15:0] = '0'::op[9:5]::coeff[9:0];
    elsif N == 32 then
        result[31:0] = '0'::op[13:6]::coeff[22:0];
    else // N == 64
        result[63:0] = '0'::op[16:6]::coeff[51:0];
    end;

    return result;
end;
```



```

// FPExpCoefficient()
// =====

func FPExpCoefficient{N}(index : integer) => bits(N)
begin
  assert N IN {16,32,64};
  var result : integer;

  if N == 16 then
    case index of
      when 0 => result = 0x000;
      when 1 => result = 0x016;
      when 2 => result = 0x02d;
      when 3 => result = 0x045;
      when 4 => result = 0x05d;
      when 5 => result = 0x075;
      when 6 => result = 0x08e;
      when 7 => result = 0x0a8;
      when 8 => result = 0x0c2;
      when 9 => result = 0x0dc;
      when 10 => result = 0x0f8;
      when 11 => result = 0x114;
      when 12 => result = 0x130;
      when 13 => result = 0x14d;
      when 14 => result = 0x16b;
      when 15 => result = 0x189;
      when 16 => result = 0x1a8;
      when 17 => result = 0x1c8;
      when 18 => result = 0x1e8;
      when 19 => result = 0x209;
      when 20 => result = 0x22b;
      when 21 => result = 0x24e;
      when 22 => result = 0x271;
      when 23 => result = 0x295;
      when 24 => result = 0x2ba;
      when 25 => result = 0x2e0;
      when 26 => result = 0x306;
      when 27 => result = 0x32e;
      when 28 => result = 0x356;
      when 29 => result = 0x37f;
      when 30 => result = 0x3a9;
      when 31 => result = 0x3d4;
    end;

  elsif N == 32 then
    case index of
      when 0 => result = 0x000000;
      when 1 => result = 0x0164d2;
      when 2 => result = 0x02cd87;
      when 3 => result = 0x043a29;
      when 4 => result = 0x05aac3;
      when 5 => result = 0x071f62;
      when 6 => result = 0x08980f;
      when 7 => result = 0x0a14d5;
      when 8 => result = 0x0b95c2;
      when 9 => result = 0x0dladf;
      when 10 => result = 0x0ea43a;
      when 11 => result = 0x1031dc;
      when 12 => result = 0x11c3d3;
      when 13 => result = 0x135a2b;
      when 14 => result = 0x14f4f0;
      when 15 => result = 0x16942d;
      when 16 => result = 0x1837f0;
      when 17 => result = 0x19e046;
      when 18 => result = 0x1b8d3a;
      when 19 => result = 0x1d3eda;
      when 20 => result = 0x1ef532;
      when 21 => result = 0x20b051;
      when 22 => result = 0x227043;
      when 23 => result = 0x243516;
      when 24 => result = 0x25fed7;
      when 25 => result = 0x27cd94;
      when 26 => result = 0x29a15b;
      when 27 => result = 0x2b7a3a;
      when 28 => result = 0x2d583f;
      when 29 => result = 0x2f3b79;
      when 30 => result = 0x3123f6;
      when 31 => result = 0x3311c4;
      when 32 => result = 0x3504f3;
      when 33 => result = 0x36fd92;
      when 34 => result = 0x38fbaf;
      when 35 => result = 0x3aff5b;
      when 36 => result = 0x3d08a4;
      when 37 => result = 0x3f179a;
      when 38 => result = 0x412c4d;
      when 39 => result = 0x4346cd;
      when 40 => result = 0x45672a;
      when 41 => result = 0x478d75;
      when 42 => result = 0x49b9be;
    end;
  end if;
end

```

```

when 43 => result = 0x4bec15;
when 44 => result = 0x4e248c;
when 45 => result = 0x506334;
when 46 => result = 0x52a81e;
when 47 => result = 0x54f35b;
when 48 => result = 0x5744fd;
when 49 => result = 0x599d16;
when 50 => result = 0x5bfbb8;
when 51 => result = 0x5e60f5;
when 52 => result = 0x60ccdf;
when 53 => result = 0x633f89;
when 54 => result = 0x65b907;
when 55 => result = 0x68396a;
when 56 => result = 0x6ac0c7;
when 57 => result = 0x6d4f30;
when 58 => result = 0x6fe4ba;
when 59 => result = 0x728177;
when 60 => result = 0x75257d;
when 61 => result = 0x77d0df;
when 62 => result = 0x7a83b3;
when 63 => result = 0x7d3e0c;
end;

else // N == 64
  case index of
    when 0 => result = 0x00000000000000;
    when 1 => result = 0x02C9A3E778061;
    when 2 => result = 0x059B0D3158574;
    when 3 => result = 0x0874518759BC8;
    when 4 => result = 0x0B5586CF9890F;
    when 5 => result = 0x0E3EC32D3D1A2;
    when 6 => result = 0x11301D0125B51;
    when 7 => result = 0x1429AAEA92DE0;
    when 8 => result = 0x172B83C7D517B;
    when 9 => result = 0x1A35BEB6FCB75;
    when 10 => result = 0x1D4873168B9AA;
    when 11 => result = 0x2063B88628CD6;
    when 12 => result = 0x2387A6E756238;
    when 13 => result = 0x26B4565E27CDD;
    when 14 => result = 0x29E9DF51FDEE1;
    when 15 => result = 0x2D285A6E4030B;
    when 16 => result = 0x306FE0A31B715;
    when 17 => result = 0x33C08B26416FF;
    when 18 => result = 0x371A7373AA9CB;
    when 19 => result = 0x3A7DB34E59FF7;
    when 20 => result = 0x3DEA64C123422;
    when 21 => result = 0x4160A21F72E2A;
    when 22 => result = 0x44E086061892D;
    when 23 => result = 0x486A2B5C13CD0;
    when 24 => result = 0x4BFDAD5362A27;
    when 25 => result = 0x4F9B2769D2CA7;
    when 26 => result = 0x5342B569D4F82;
    when 27 => result = 0x56F4736B527DA;
    when 28 => result = 0x5AB07DD485429;
    when 29 => result = 0x5E76F15AD2148;
    when 30 => result = 0x6247EB03A5585;
    when 31 => result = 0x6623882552225;
    when 32 => result = 0x6A09E667F3BCD;
    when 33 => result = 0x6DFB23C651A2F;
    when 34 => result = 0x71F75E8EC5F74;
    when 35 => result = 0x75FEB564267C9;
    when 36 => result = 0x7A11473EB0187;
    when 37 => result = 0x7E2F336CF4E62;
    when 38 => result = 0x82589994CCE13;
    when 39 => result = 0x868D99B4492ED;
    when 40 => result = 0x8ACE5422AA0DB;
    when 41 => result = 0x8F1AE99157736;
    when 42 => result = 0x93737B0CDC5E5;
    when 43 => result = 0x97D829FDE4E50;
    when 44 => result = 0x9C49182A3F090;
    when 45 => result = 0xA0C667B5DE565;
    when 46 => result = 0xA5503B23E255D;
    when 47 => result = 0xA9E6B5579FDBF;
    when 48 => result = 0xAE89F995AD3AD;
    when 49 => result = 0xB33A2B84F15FB;
    when 50 => result = 0xB7F76F2FB5E47;
    when 51 => result = 0xBCC1E904BC1D2;
    when 52 => result = 0xC199BDD85529C;
    when 53 => result = 0xC67F12E57D14B;
    when 54 => result = 0xCB720DCEF9069;
    when 55 => result = 0xD072D4A07897C;
    when 56 => result = 0xD5818DCFB4487;
    when 57 => result = 0xDA9E603DB3285;
    when 58 => result = 0xDFC9737B9B5F;
    when 59 => result = 0xE502EE78B3FF6;
    when 60 => result = 0xEA4AFA2A490DA;
    when 61 => result = 0xEFA1BEE615A27;
    when 62 => result = 0xF50765B6E4540;
    when 63 => result = 0xFA7C1819E90D8;
  end;
end;

```

```

    end;
end;

return result[N-1:0];
end;

```

Library pseudocode for aarch64/functions/sve/FPLogB

```

// FPLogB()
// =====

func FPLogB{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var result : integer;
    var (fptype,sign,value) : (FType, bit, real) = FPUnpack{N}(op, fpcr);

    if fptype == FType_SNaN || fptype == FType_QNaN || fptype == FType_Zero then
        FPProcessException(FPExc_InvalidOp, fpcr);
        result = -(2^(N-1)); // MinInt, 100..00
    elsif fptype == FType_Infinity then
        result = 2^(N-1) - 1; // MaxInt, 011..11
    else
        // FPUnpack has already scaled a subnormal input
        value = Abs(value);
        (value, result) = NormalizeReal(value);

        FPProcessDenorm(fptype, N, fpcr);
    end;
    return result[N-1:0];
end;

```

Library pseudocode for aarch64/functions/sve/FPMinNormal

```

// FPMinNormal()
// =====

func FPMinNormal{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = Zeros{E-1}::'1';
    let frac : bits(F) = Zeros{};
    return sign :: exp :: frac;
end;

```

Library pseudocode for aarch64/functions/sve/FPOne

```

// FPOne()
// =====

func FPOne{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = '0'::Ones{E-1};
    let frac : bits(F) = Zeros{};
    return sign :: exp :: frac;
end;

```

Library pseudocode for aarch64/functions/sve/FPPointFive

```

// FPPointFive()
// =====

func FPPointFive{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = '0'::Ones{E-2}::'0';
    let frac : bits(F) = Zeros{};
    return sign :: exp :: frac;
end;

```

Library pseudocode for aarch64/functions/sve/FPReducePredicated

```
// FPReducePredicated()
// =====
func FPReducePredicated{esize, N, M}(op : ReduceOp, input : bits(N), mask : bits(M),
                                     identity : bits(esize), fpcr : FPCR_Type) => bits(esize)
begin
    assert(N == M * 8);
    assert(IsPow2(N));
    var operand : bits(N);
    let elements : integer = N DIV esize;

    for e = 0 to elements-1 do
        if e * esize < N && ActivePredicateElement{M}(mask, e, esize) then
            operand[e*esize] = input[e*esize];
        else
            operand[e*esize] = identity;
        end;
    end;

    return FPReduce{esize, N}(op, operand, fpcr);
end;
```

Library pseudocode for aarch64/functions/sve/FPTrigMAdd

```
// FPTrigMAdd()
// =====
func FPTrigMAdd{N}(x_in : integer, op1 : bits(N), op2_in : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var coeff : bits(N);
    var op2 : bits(N) = op2_in;
    var x : integer = x_in;
    assert x >= 0;
    assert x < 8;

    if op2[N-1] == '1' then
        x = x + 8;
    end;

    coeff = FPTrigMAddCoefficient{N}(x);
    // Safer to use EffectiveFPCR() in case the input fpcr argument
    // is modified as opposed to actual value of FPCR

    op2 = FPAbs{N}(op2, EffectiveFPCR());
    let result : bits(N) = FPMulAdd{}(coeff, op1, op2, fpcr);
    return result;
end;
```

Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =====

func FPTrigMAddCoefficient{N}(index : integer) => bits(N)
begin
  assert N IN {16,32,64};
  var result : integer;

  if N == 16 then
    case index of
      when 0 => result = 0x3c00;
      when 1 => result = 0xb155;
      when 2 => result = 0x2030;
      when 3 => result = 0x0000;
      when 4 => result = 0x0000;
      when 5 => result = 0x0000;
      when 6 => result = 0x0000;
      when 7 => result = 0x0000;
      when 8 => result = 0x3c00;
      when 9 => result = 0xb800;
      when 10 => result = 0x293a;
      when 11 => result = 0x0000;
      when 12 => result = 0x0000;
      when 13 => result = 0x0000;
      when 14 => result = 0x0000;
      when 15 => result = 0x0000;
    end;
  elsif N == 32 then
    case index of
      when 0 => result = 0x3f800000;
      when 1 => result = 0xbe2aaaaab;
      when 2 => result = 0x3c088886;
      when 3 => result = 0xb95008b9;
      when 4 => result = 0x36369d6d;
      when 5 => result = 0x00000000;
      when 6 => result = 0x00000000;
      when 7 => result = 0x00000000;
      when 8 => result = 0x3f800000;
      when 9 => result = 0xbf000000;
      when 10 => result = 0x3d2aaaaa6;
      when 11 => result = 0xbab60705;
      when 12 => result = 0x37cd37cc;
      when 13 => result = 0x00000000;
      when 14 => result = 0x00000000;
      when 15 => result = 0x00000000;
    end;
  else // N == 64
    case index of
      when 0 => result = 0x3ff0000000000000;
      when 1 => result = 0xbfc5555555555543;
      when 2 => result = 0x3f8111111110f30c;
      when 3 => result = 0xbf2a01a019b92fc6;
      when 4 => result = 0x3ec71de351f3d22b;
      when 5 => result = 0xbe5ae5e2b60f7b91;
      when 6 => result = 0x3de5d8408868552f;
      when 7 => result = 0x0000000000000000;
      when 8 => result = 0x3ff0000000000000;
      when 9 => result = 0xbfe0000000000000;
      when 10 => result = 0x3fa5555555555536;
      when 11 => result = 0xbf56c16c16c13a0b;
      when 12 => result = 0x3efa01a019b1e8d8;
      when 13 => result = 0xbe927e4f7282f468;
      when 14 => result = 0x3e21ee96d2641b13;
      when 15 => result = 0xbda8f76380fbb401;
    end;
  end;

  return result[N-1:0];
end;
```


Library pseudocode for aarch64/functions/sve/FPTrigSMul

```
// FPTrigSMul()
// =====

func FPTrigSMul{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var result : bits(N) = FPMul{}(op1, op1, fpcr);
    let fpexc = FALSE;
    let (fptype, sign, value) : (FType, bit, real) = FPUnpack{N}(result, fpcr, fpexc);

    if ! fptype IN {FType_QNaN, FType_SNaN} then
        result[N-1] = op2[0];
    end;

    return result;
end;
```

Library pseudocode for aarch64/functions/sve/FPTrigSSel

```
// FPTrigSSel()
// =====

func FPTrigSSel{N}(op1 : bits(N), op2 : bits(N)) => bits(N)
begin
    assert N IN {16,32,64};
    var result : bits(N);

    if op2[0] == '1' then
        result = FPOne{N}(op2[1]);
    elsif op2[1] == '1' then
        result = FPNeg{N}(op1, EffectiveFPCR());
    else
        result = op1;
    end;

    return result;
end;
```

Library pseudocode for aarch64/functions/sve/FirstActive

```
// FirstActive()
// =====

func FirstActive{N}(mask : bits(N), x : bits(N), esize : integer) => bit
begin
    let elements : integer = N DIV (esize DIV 8);
    for e = 0 to elements-1 do
        if ActivePredicateElement{N}(mask, e, esize) then
            return PredicateElement{N}(x, e, esize);
        end;
    end;
    return '0';
end;
```

Library pseudocode for aarch64/functions/sve/HaveSVE2FP8DOT2

```
// HaveSVE2FP8DOT2()
// =====
// Returns TRUE if SVE2 FP8 dot product to half-precision instructions
// are implemented, FALSE otherwise.

func HaveSVE2FP8DOT2() => boolean
begin
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8DOT2)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8DOT2));
end;
```

Library pseudocode for aarch64/functions/sve/HaveSVE2FP8DOT4

```
// HaveSVE2FP8DOT4()
// =====
// Returns TRUE if SVE2 FP8 dot product to single-precision instructions
// are implemented, FALSE otherwise.

func HaveSVE2FP8DOT4() => boolean
begin
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8DOT4)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8DOT4));
end;
```

Library pseudocode for aarch64/functions/sve/HaveSVE2FP8FMA

```
// HaveSVE2FP8FMA()
// =====
// Returns TRUE if SVE2 FP8 multiply-accumulate to half-precision and single-precision
// instructions are implemented, FALSE otherwise.

func HaveSVE2FP8FMA() => boolean
begin
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8FMA)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8FMA));
end;
```

Library pseudocode for aarch64/functions/sve/ImplementedSMEVectorLength

```
// ImplementedSMEVectorLength()
// =====
// Reduce SVE/SME vector length to a supported value (power of two)

readonly func ImplementedSMEVectorLength(nbits_in : integer) => VecLen
begin
    let maxbits : VecLen = MaxImplementedSVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    var nbits : integer = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && AlignDownSize(nbits, 128) == nbits;

    // Search for a supported power-of-two VL less than or equal to nbits
    while nbits > 128 && !SupportedPowerTwoSVL(nbits) looplimit 7 do
        nbits = nbits - 128;
    end;

    // Return the smallest supported power-of-two VL
    while nbits < maxbits && !SupportedPowerTwoSVL(nbits) looplimit 7 do
        nbits = nbits * 2;
    end;

    return nbits as VecLen;
end;
```

Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// =====
// Reduce SVE vector length to a supported value (power of two)

readonly func ImplementedSVEVectorLength(nbits_in : integer) => VecLen
begin
    let maxbits : integer = MaxImplementedVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    var nbits : integer = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && AlignDownSize(nbits, 128) == nbits;

    while !IsPow2(nbits) looplimit 7 do
        nbits = nbits - 128;
    end;
    return nbits as VecLen;
end;
```

Library pseudocode for aarch64/functions/sve/InStreamingMode

```
// InStreamingMode()
// =====

func InStreamingMode() => boolean
begin
    return IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1';
end;
```

Library pseudocode for aarch64/functions/sve/IntReducePredicated

```
// IntReducePredicated()
// =====
func IntReducePredicated(esize, N, M)(op : ReduceOp, input : bits(N),
                                     mask : bits(M), identity : bits(esize)) => bits(esize)
begin
    assert(N == M * 8);
    assert IsPow2(N);
    var operand : bits(N);
    let elements : integer = N DIV esize;

    for e = 0 to elements-1 do
        if e * esize < N && ActivePredicateElement{M}(mask, e, esize) then
            operand[e*:esize] = input[e*:esize];
        else
            operand[e*:esize] = identity;
        end;
    end;

    return IntReduce(esize, N)(op, operand);
end;
```

Library pseudocode for aarch64/functions/sve/IsFPEnabled

```
// IsFPEnabled()
// =====
// Returns TRUE if accesses to the Advanced SIMD and floating-point
// registers are enabled at the target exception level in the current
// execution state and FALSE otherwise.

func IsFPEnabled(el : bits(2)) => boolean
begin
    if ELUsingAArch32(el) then
        return AArch32 IsFPEnabled(el);
    else
        return AArch64 IsFPEnabled(el);
    end;
end;
```

Library pseudocode for aarch64/functions/sve/IsFullA64Enabled

```
// IsFullA64Enabled()
// =====
// Returns TRUE if full A64 is enabled in Streaming mode and FALSE otherwise.

func IsFullA64Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_SME_FA64) then return FALSE; end;

    // Check if full A64 disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check full A64 at EL0/EL1
        if SMCR\_EL1().FA64 == '0' then return FALSE; end;
    end;

    // Check if full A64 disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR\_EL2().FA64 == '0' then return FALSE; end;
    end;

    // Check if full A64 disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR\_EL3().FA64 == '0' then return FALSE; end;
    end;

    return TRUE;
end;
```

Library pseudocode for aarch64/functions/sve/IsOriginalSVEEnabled

```
// IsOriginalSVEEnabled()
// =====
// Returns TRUE if access to SVE functionality is enabled at the target
// exception level and FALSE otherwise.

func IsOriginalSVEEnabled(el : bits(2)) => boolean
begin
    var disabled : boolean;
    if ELUsingAArch32(el) then
        return FALSE;
    end;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1().ZEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = el == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then return FALSE; end;
    end;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            case CPTR_EL2().ZEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = el == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then return FALSE; end;
        else
            if CPTR_EL2().TZ == '1' then return FALSE; end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().EZ == '0' then return FALSE; end;
    end;

    return TRUE;
end;
```

Library pseudocode for aarch64/functions/sve/IsSMEEnabled

```
// IsSMEEnabled()
// =====
// Returns TRUE if access to SME functionality is enabled at the target
// exception level and FALSE otherwise.

func IsSMEEnabled(el : bits(2)) => boolean
begin
    var disabled : boolean;
    if ELUsingAArch32(el) then
        return FALSE;
    end;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1().SMEN of
            when 'x0' => disabled = TRUE;
            when '01' => disabled = el == EL0;
            when '11' => disabled = FALSE;
        end;
        if disabled then return FALSE; end;
    end;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            case CPTR_EL2().SMEN of
                when 'x0' => disabled = TRUE;
                when '01' => disabled = el == EL0 && HCR_EL2().TGE == '1';
                when '11' => disabled = FALSE;
            end;
            if disabled then return FALSE; end;
        else
            if CPTR_EL2().TSM == '1' then return FALSE; end;
        end;
    end;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3().ESM == '0' then return FALSE; end;
    end;

    return TRUE;
end;
```

Library pseudocode for aarch64/functions/sve/IsSVEEnabled

```
// IsSVEEnabled()
// =====
// Returns TRUE if access to SVE registers is enabled at the target exception
// level and FALSE otherwise.

func IsSVEEnabled(el : bits(2)) => boolean
begin
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        return IsSMEEnabled(el);
    elsif IsFeatureImplemented(FEAT_SVE) then
        return IsOriginalSVEEnabled(el);
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch64/functions/sve/LastActive

```
// LastActive()
// =====

func LastActive{N}(mask : bits(N), x : bits(N), esize : integer) => bit
begin
    let elements : integer = N DIV (esize DIV 8);
    for e = elements-1 downto 0 do
        if ActivePredicateElement{N}(mask, e, esize) then
            return PredicateElement{N}(x, e, esize);
        end;
    end;
    return '0';
end;
```

Library pseudocode for aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// =====

func LastActiveElement{N}(mask : bits(N), esize : integer) => integer
begin
    let elements : integer = N DIV (esize DIV 8);
    for e = elements-1 downto 0 do
        if ActivePredicateElement{N}(mask, e, esize) then return e; end;
    end;
    return -1;
end;
```

Library pseudocode for aarch64/functions/sve/MaxImplementedAnyVL

```
// MaxImplementedAnyVL()
// =====

readonly func MaxImplementedAnyVL() => integer
begin
    if IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_SVE) then
        return Max(MaxImplementedVL(), MaxImplementedSVL());
    end;
    if IsFeatureImplemented(FEAT_SME) then
        return MaxImplementedSVL();
    end;
    return MaxImplementedVL();
end;
```

Library pseudocode for aarch64/functions/sve/MaxImplementedSVL

```
// MaxImplementedSVL()
// =====

readonly func MaxImplementedSVL() => VecLen
begin
    return ImpDefInt("Max implemented SVL") as VecLen;
end;
```

Library pseudocode for aarch64/functions/sve/MaxImplementedVL

```
// MaxImplementedVL()
// =====

readonly func MaxImplementedVL() => integer
begin
    return ImpDefInt("Max implemented VL");
end;
```

Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// =====

func MaybeZeroSVEUppers(target_el : bits(2))
begin
    var lower_enabled : boolean;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;
    end;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEEnabled(EL2);
        else
            lower_enabled = IsFPEEnabled(EL1);
        end;
    elseif target_el == EL2 then
        assert EL2Enabled() && !ELUsingAArch32(EL2);
        if HCR_EL2().TGE == '0' then
            lower_enabled = IsFPEEnabled(EL1);
        else
            lower_enabled = IsFPEEnabled(EL0);
        end;
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEEnabled(EL0);
    end;

    if lower_enabled then
        let VL : integer{} = if IsSVEEnabled(PSTATE.EL) then CurrentVL() else 128;
        let PL : integer{} = VL DIV 8;
        for n = 0 to 31 do
            if ConstrainUnpredictableBool(Unpredictable SVEZEROUPPER) then
                Z[[n]] = ZeroExtend{MAX_VL}(Z[[n]][VL-1:0]);
            end;
        end;
        for n = 0 to 15 do
            if ConstrainUnpredictableBool(Unpredictable SVEZEROUPPER) then
                P[[n]] = ZeroExtend{MAX_PL}(P[[n]][PL-1:0]);
            end;
        end;
        if ConstrainUnpredictableBool(Unpredictable SVEZEROUPPER) then
            FFR = ZeroExtend{MAX_PL}(FFR[PL-1:0]);
        end;
        if IsFeatureImplemented(FEAT_SME) && PSTATE.ZA == '1' then
            let SVL : integer{} = CurrentSVL();
            let accessiblevecs : integer = SVL DIV 8;
            let allvecs : integer = MaxImplementedSVL() DIV 8;

            for n = 0 to accessiblevecs - 1 do
                if ConstrainUnpredictableBool(Unpredictable SMEZEROUPPER) then
                    ZA[[n]] = ZeroExtend{MAX_VL}(ZA[[n]][SVL-1:0]);
                end;
            end;
            for n = accessiblevecs to allvecs - 1 do
                if ConstrainUnpredictableBool(Unpredictable SMEZEROUPPER) then
                    ZA[[n]] = Zeros{MAX_VL};
                end;
            end;
        end;
    end;
end;
end;
```

Library pseudocode for aarch64/functions/sve/MemNF

```
// MemNF
// =====

func MemNF{size : integer{8, 16, 32, 64, 128}}(address : bits(64),
                                              accdesc : AccessDescriptor
                                              ) => (bits(size), boolean)

begin
  let bytes : integer{} = size DIV 8;
  var value : bits(size);
  var bad : boolean;

  var aligned : boolean = IsAlignedSize{64}(address, bytes);

  if !aligned && AlignmentEnforced() then
    return (ARBITRARY : bits(size), TRUE);
  end;

  let atomic : boolean = aligned || bytes == 1;

  if !atomic then
    (value[7:0], bad) = MemSingleNF{8}(address, accdesc, aligned);

    if bad then
      return (ARBITRARY : bits(size), TRUE);
    end;

    // For subsequent bytes, if they cross to a new translation page which assigns
    // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
    // will generate an Alignment Fault.
    if !aligned then
      let c : Constraint = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
      assert c IN {Constraint\_FAULT, Constraint\_NONE};
      if c == Constraint\_NONE then aligned = TRUE; end;
    end;

    for i = 1 to bytes-1 do
      (value[i*8], bad) = MemSingleNF{8}(address+i, accdesc, aligned);

      if bad then
        return (ARBITRARY : bits(size), TRUE);
      end;
    end;
  else
    (value, bad) = MemSingleNF{size}(address, accdesc, aligned);
    if bad then
      return (ARBITRARY : bits(size), TRUE);
    end;
  end;

  if BigEndian(accdesc.acctype) then
    value = BigEndianReverse{size}(value);
  end;

  return (value, FALSE);
end;
```


Library pseudocode for aarch64/functions/sve/MemSingleNF

```
// MemSingleNF
// =====

func MemSingleNF(size : integer{8, 16, 32, 64, 128})(address : bits(64),
    accdesc_in : AccessDescriptor,
    aligned : boolean) => (bits(size), boolean)

begin
    assert accdesc_in.acctype == AccessType\_SVE;
    assert accdesc_in.nonfault || (accdesc_in.firstfault && !accdesc_in.first);

    let bytes : integer{} = size DIV 8;
    var value : bits(size);
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;
    var accdesc : AccessDescriptor = accdesc_in;
    var fault : FaultRecord = NoFault(accdesc, address);

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool(Unpredictable\_NONFAULT) then
        return (ARBITRARY : bits(size), TRUE);
    end;

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64\_AccessIsTagChecked(address, accdesc);
    end;

    // MMU or MPU
    memaddrdesc = AArch64\_TranslateAddress(address, accdesc, aligned, bytes);

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.memattrs.memtype == MemType\_Device then
        return (ARBITRARY : bits(size), TRUE);
    end;

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (ARBITRARY : bits(size), TRUE);
    end;

    if accdesc.tagchecked then
        let ltag : bits(4) = AArch64\_LogicalAddressTag(address);
        let readcheck : boolean = TRUE;
        fault = AArch64\_CheckTag(memaddrdesc, accdesc, readcheck, bytes, ltag);
        if fault.statuscode != Fault\_None then
            return (ARBITRARY : bits(size), TRUE);
        end;
    end;

    (memstatus, value) = PhysMemRead(size)(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        let iswrite : boolean = FALSE;
        if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc, bytes, accdesc) then
            return (ARBITRARY : bits(size), TRUE);
        end;
        fault.merrorstate = memstatus.merrorstate;
        fault.extflag = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        PendSErrorInterrupt(fault);
    end;

    return (value, FALSE);
end;
```

Library pseudocode for aarch64/functions/sve/NoneActive

```
// NoneActive()
// =====

func NoneActive(N)(mask : bits(N), x : bits(N), esize : integer) => bit

begin
    let elements : integer = N DIV (esize DIV 8);
    for e = 0 to elements-1 do
        if ActivePredicateElement{N}(mask, e, esize) && ActivePredicateElement{N}(x, e, esize) then
            return '0';
        end;
    end;
    return '1';
end;
```

Library pseudocode for aarch64/functions/sve/P

```
// P - accessor
// =====

accessor P{width}(n : integer) <=> value : bits(width)
begin
  getter
    assert n >= 0 && n <= 31;
    assert width == CurrentVL() DIV 8;
    return \_P[[n]][width-1:0];
  end;

  setter
    assert n >= 0 && n <= 31;
    assert width == CurrentVL() DIV 8;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZERoupper) then
      \_P[[n]] = ZeroExtend(MAX\_PL)(value);
    else
      \_P[[n]][width-1:0] = value;
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/sve/PredLen

```
// PredLen
// =====

type PredLen of integer{16, 32, 64, 128, 256};
```

Library pseudocode for aarch64/functions/sve/PredTest

```
// PredTest()
// =====

func PredTest{N}(mask : bits(N), result : bits(N), esize : integer) => bits(4)
begin
  let n : bit = FirstActive{N}(mask, result, esize);
  let z : bit = NoneActive{N}(mask, result, esize);
  let c : bit = NOT LastActive{N}(mask, result, esize);
  let v : bit = '0';
  return n::z::c::v;
end;
```

Library pseudocode for aarch64/functions/sve/PredicateElement

```
// PredicateElement()
// =====
// Returns the predicate bit

func PredicateElement{N}(pred : bits(N), e : integer, esize : integer) => bit
begin
  assert esize IN {8, 16, 32, 64, 128};
  let n : integer = e * (esize DIV 8);
  assert n >= 0 && n < N;
  return pred[n];
end;
```

Library pseudocode for aarch64/functions/sve/ResetSMEState

```
// ResetSMEState()
// =====

func ResetSMEState(newenable : bit)
begin
  let vectors : integer = MAX\_VL DIV 8;
  if newenable == '1' then
    for n = 0 to vectors - 1 do
      \_ZA[[n]] = Zeros(MAX\_VL);
    end;
    if IsFeatureImplemented(FEAT_SME2) then
      \_ZT0 = Zeros(ZT0\_LEN);
    end;
  else
    for n = 0 to vectors - 1 do
      \_ZA[[n]] = ARBITRARY : bits(MAX\_VL);
    end;
    if IsFeatureImplemented(FEAT_SME2) then
      \_ZT0 = ARBITRARY : bits(ZT0\_LEN);
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/sve/ResetSVERegisters

```
// ResetSVERegisters()
// =====

func ResetSVERegisters()
begin
  for n = 0 to 31 do
    Z[[n]] = ARBITRARY : bits(MAX_VL);
  end;
  for n = 0 to 15 do
    P[[n]] = ARBITRARY : bits(MAX_PL);
  end;
  FFR = ARBITRARY : bits(MAX_PL);
end;
```

Library pseudocode for aarch64/functions/sve/ResetSVEState

```
// ResetSVEState()
// =====

func ResetSVEState()
begin
  for n = 0 to 31 do
    Z[[n]] = Zeros(MAX_VL);
  end;
  for n = 0 to 15 do
    P[[n]] = Zeros(MAX_PL);
  end;
  FFR = Zeros(MAX_PL);
  FPSR() = ZeroExtend{64}(0x0800009f[31:0]);
  FPMR() = Zeros{64};
end;
```

Library pseudocode for aarch64/functions/sve/SMEAccessTrap

```
// SMEAccessTrap()
// =====
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

func SMEAccessTrap(etype : SMEExceptionType, target_el_in : bits(2))
begin
  var target_el : bits(2) = target_el_in;
  assert UInt(target_el) >= UInt(PSTATE.EL);
  if target_el == EL0 then
    target_el = EL1;
  end;
  let route_to_el2 : boolean = (PSTATE.EL == EL0 && target_el == EL1 &&
    EL2Enabled() && HCR_EL2().TGE == '1');

  var except : ExceptionRecord = ExceptionSyndrome(Exception SMEAccessTrap);
  let preferred_exception_return : bits(64) = ThisInstrAddr{}();
  let vect_offset : integer = 0x0;

  case etype of
    when SMEExceptionType AccessTrap =>
      except.syndrome.iss[2:0] = '000';
    when SMEExceptionType Streaming =>
      except.syndrome.iss[2:0] = '001';
    when SMEExceptionType NotStreaming =>
      except.syndrome.iss[2:0] = '010';
    when SMEExceptionType InactiveZA =>
      except.syndrome.iss[2:0] = '011';
    when SMEExceptionType InaccessibleZT0 =>
      except.syndrome.iss[2:0] = '100';
  end;

  if route_to_el2 then
    AArch64_TakeException(EL2, except, preferred_exception_return, vect_offset);
  else
    AArch64_TakeException(target_el, except, preferred_exception_return, vect_offset);
  end;
end;
```

Library pseudocode for aarch64/functions/sve/SMEExceptionType

```
// SMEExceptionType
// =====
type SMEExceptionType of enumeration {
  SMEExceptionType_AccessTrap, // SME functionality trapped or disabled
  SMEExceptionType_Streaming, // Illegal instruction in Streaming SVE mode
  SMEExceptionType_NotStreaming, // Illegal instruction not in Streaming SVE mode
  SMEExceptionType_InactiveZA, // Illegal instruction when ZA is inactive
  SMEExceptionType_InaccessibleZT0, // Access to ZT0 is disabled
};
```

Library pseudocode for aarch64/functions/sve/SVCR

```
// SVCR
// ====

accessor SVCR() <=> value : SVCR_Type
begin
    // Returns PSTATE.[ZA, SM]
    getter
        let value : SVCR_Type = Zeros{62} :: PSTATE.ZA :: PSTATE.SM;
        return value;
    end;

    // Sets PSTATE.[ZA, SM]
    setter
        SetPSTATE\_SM(value[0]);
        SetPSTATE\_ZA(value[1]);
        return;
    end;
end;
```

Library pseudocode for aarch64/functions/sve/SVEAccessTrap

```
// SVEAccessTrap()
// =====
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

func SVEAccessTrap(target_el : bits(2))
begin
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    let route_to_el2 : boolean = target_el == EL1 && EL2Enabled() && HCR\_EL2().TGE == '1';

    let except : ExceptionRecord = ExceptionSyndrome(Exception\_SVEAccessTrap);
    let preferred_exception_return : bits(64) = ThisInstrAddr()();
    let vect_offset : integer{} = 0x0;

    if route_to_el2 then
        AArch64\_TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64\_TakeException(target_el, except, preferred_exception_return, vect_offset);
    end;
end;
```

Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

```
// SVEMoveMaskPreferred()
// =====
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

func SVEMoveMaskPreferred(imm13 : bits(13)) => boolean
begin
    var imm : bits(64);
    (imm, -) = DecodeBitMasks{64}(imm13[12], imm13[5:0], imm13[11:6], TRUE);

    // Check for 8 bit immediates
    if !IsZero(imm[7:0]) then
        // Check for 'ffffffffffffxy' or '00000000000000xy'
        if IsZero(imm[63:7]) || IsOnes(imm[63:7]) then
            return FALSE;
        end;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm[63:32] == imm[31:0] && (IsZero(imm[31:7]) || IsOnes(imm[31:7])) then
            return FALSE;
        end;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if (imm[63:32] == imm[31:0] && imm[31:16] == imm[15:0] &&
            (IsZero(imm[15:7]) || IsOnes(imm[15:7]))) then
            return FALSE;
        end;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm[63:32] == imm[31:0] && imm[31:16] == imm[15:0] && (imm[15:8] == imm[7:0]) then
            return FALSE;
        end;

    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '00000000000000xy00'
        if IsZero(imm[63:15]) || IsOnes(imm[63:15]) then
            return FALSE;
        end;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm[63:32] == imm[31:0] && (IsZero(imm[31:7]) || IsOnes(imm[31:7])) then
            return FALSE;
        end;

        // Check for 'xy00xy00xy00xy00'
        if imm[63:32] == imm[31:0] && imm[31:16] == imm[15:0] then
            return FALSE;
        end;
    end;

    return TRUE;
end;
```

Library pseudocode for aarch64/functions/sve/SetPSTATE_SM

```
// SetPSTATE_SM()
// =====

func SetPSTATE_SM(value : bit)
begin
    if PSTATE.SM != value then
        ResetSVEState();
        PSTATE.SM = value;
    end;
end;
```

Library pseudocode for aarch64/functions/sve/SetPSTATE_ZA

```
// SetPSTATE_ZA()
// =====

func SetPSTATE_ZA(value : bit)
begin
    if PSTATE.ZA != value then
        ResetSMEState(value);
        PSTATE.ZA = value;
    end;
end;
```

Library pseudocode for aarch64/functions/sve/SupportedPowerTwoSVL

```
// SupportedPowerTwoSVL()
// =====
// Return an IMPLEMENTATION DEFINED specific value
// returns TRUE if SVL is supported and is a power of two, FALSE otherwise

readonly impdef func SupportedPowerTwoSVL(nbits : integer) => boolean
begin
    return IsPow2(nbits) && (nbits <= MAX\_VL);
end;
```

Library pseudocode for aarch64/functions/sve/System

```
// System Registers
// =====

constant MAX_VL : VecLen = 2048;
constant MAX_PL : PredLen = 256;
constant ZT0_LEN : integer{} = 512;
var _Z : array [[32]] of bits(MAX\_VL);
var _P : array [[16]] of bits(MAX\_PL);
var _FFR : bits(MAX\_PL);
```

Library pseudocode for aarch64/functions/sve/VecLen

```
// VecLen
// =====

type VecLen of integer{128, 256, 512, 1024, 2048};
```

Library pseudocode for aarch64/functions/sve/Z

```
// Z - accessor
// =====

accessor Z{width}(n : integer) <=> value : bits(width)
begin
    getter
        assert n >= 0 && n <= 31;
        assert width == CurrentVL();
        return \_Z[[n]][width-1:0];
    end;
    setter
        assert n >= 0 && n <= 31;
        assert width == CurrentVL();
        if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
            \_Z[[n]] = ZeroExtend(MAX\_VL)(value);
        else
            \_Z[[n]][width-1:0] = value;
        end;
    end;
end;
```

Library pseudocode for aarch64/functions/syshintop/SystemHintOp

```
// SystemHintOp
// =====
// System Hint instruction types.

type SystemHintOp of enumeration {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_WFET,
    SystemHintOp_WFIT,
    SystemHintOp_CLRBHB,
    SystemHintOp_GCSB,
    SystemHintOp_CHKFEAT,
    SystemHintOp_STSHH,
    SystemHintOp_CSDB
};
```

Library pseudocode for aarch64/functions/syslop/SysLOp

```
// SysLOp()
// =====

func SysLOp(op1 : bits(3), CRn : bits(4), CRm : bits(4), op2 : bits(3)) => SystemLOp
begin
    return Sysl\_SYSL;
end;
```

Library pseudocode for aarch64/functions/syslop/SystemLOp

```
// SystemLOp
// =====
// System instruction with result types.

type SystemLOp of enumeration {
    Sysl_SYSL
};
```



```

// SysOp()
// =====

func SysOp(op1 : bits(3), CRn : bits(4), CRm : bits(4), op2 : bits(3)) => SystemOp
begin
  case op1::CRn::CRm::op2 of
    when '000 0111 1000 000' => return Sys AT;      // S1E1R
    when '000 0111 1000 001' => return Sys AT;      // S1E1W
    when '000 0111 1000 010' => return Sys AT;      // S1E0R
    when '000 0111 1000 011' => return Sys AT;      // S1E0W
    when '000 0111 1001 000' => return Sys AT;      // S1E1RP
    when '000 0111 1001 001' => return Sys AT;      // S1E1WP
    when '000 0111 1001 010' => return Sys AT;      // S1E1A
    when '100 0111 1000 000' => return Sys AT;      // S1E2R
    when '100 0111 1000 001' => return Sys AT;      // S1E2W
    when '100 0111 1001 010' => return Sys AT;      // S1E2A
    when '100 0111 1000 100' => return Sys AT;      // S1E21R
    when '100 0111 1000 101' => return Sys AT;      // S1E21W
    when '100 0111 1000 110' => return Sys AT;      // S1E20R
    when '100 0111 1000 111' => return Sys AT;      // S1E20W
    when '110 0111 1000 000' => return Sys AT;      // S1E3R
    when '110 0111 1000 001' => return Sys AT;      // S1E3W
    when '110 0111 1001 010' => return Sys AT;      // S1E3A
    when '001 0111 0010 100' => return Sys BRB;     // IALL
    when '001 0111 0010 101' => return Sys BRB;     // INJ
    when '000 0111 0110 001' => return Sys DC;      // IVAC
    when '000 0111 0110 010' => return Sys DC;      // ISW
    when '000 0111 0110 011' => return Sys DC;      // IGvac
    when '000 0111 0110 100' => return Sys DC;      // IGSW
    when '000 0111 0110 101' => return Sys DC;      // IGDvac
    when '000 0111 0110 110' => return Sys DC;      // IGDSW
    when '000 0111 1010 010' => return Sys DC;      // CSW
    when '000 0111 1010 100' => return Sys DC;      // CGSW
    when '000 0111 1010 110' => return Sys DC;      // CGDSW
    when '000 0111 1110 010' => return Sys DC;      // CISW
    when '000 0111 1110 100' => return Sys DC;      // CIGSW
    when '000 0111 1110 110' => return Sys DC;      // CIGDSW
    when '011 0111 0100 001' => return Sys DC;      // ZVA
    when '011 0111 0100 011' => return Sys DC;      // GVA
    when '011 0111 0100 100' => return Sys DC;      // GZVA
    when '011 0111 1010 001' => return Sys DC;      // CVAC
    when '011 0111 1010 011' => return Sys DC;      // CGvac
    when '011 0111 1010 101' => return Sys DC;      // CGDvac
    when '011 0111 1011 001' => return Sys DC;      // CVAU
    when '011 0111 1100 001' => return Sys DC;      // CVAP
    when '011 0111 1100 011' => return Sys DC;      // CGVAP
    when '011 0111 1100 101' => return Sys DC;      // CGDVAP
    when '011 0111 1101 001' => return Sys DC;      // CVADP
    when '011 0111 1101 011' => return Sys DC;      // CGVADP
    when '011 0111 1101 101' => return Sys DC;      // CGDVADP
    when '011 0111 1110 001' => return Sys DC;      // CIVAC
    when '011 0111 1110 011' => return Sys DC;      // CIGvac
    when '011 0111 1110 101' => return Sys DC;      // CIGDvac
    when '100 0111 1110 000' => return Sys DC;      // CIPAE
    when '100 0111 1110 111' => return Sys DC;      // CIGDPAE
    when '110 0111 1110 001' => return Sys DC;      // CIPAPA
    when '110 0111 1110 101' => return Sys DC;      // CIGDPAPA
    when '000 0111 1111 001' => return Sys DC;      // CIVAPS
    when '000 0111 1111 101' => return Sys DC;      // CIGDVAPS
    when '000 0111 0001 000' => return Sys IC;      // IALLUIS
    when '000 0111 0101 000' => return Sys IC;      // IALLU
    when '011 0111 0101 001' => return Sys IC;      // IVAU
    when '000 1000 0001 000' => return Sys TLBI;     // VMALLE10S
    when '000 1000 0001 001' => return Sys TLBI;     // VAE10S
    when '000 1000 0001 010' => return Sys TLBI;     // ASIDE10S
    when '000 1000 0001 011' => return Sys TLBI;     // VAAE10S
    when '000 1000 0001 101' => return Sys TLBI;     // VALE10S
    when '000 1000 0001 111' => return Sys TLBI;     // VAALE10S
    when '000 1000 0010 001' => return Sys TLBI;     // RVAE11S
    when '000 1000 0010 011' => return Sys TLBI;     // RVAAE11S
    when '000 1000 0010 101' => return Sys TLBI;     // RVALE11S
    when '000 1000 0010 111' => return Sys TLBI;     // RVAALE11S
    when '000 1000 0011 000' => return Sys TLBI;     // VMALLE11S
    when '000 1000 0011 001' => return Sys TLBI;     // VAE11S
    when '000 1000 0011 010' => return Sys TLBI;     // ASIDE11S
    when '000 1000 0011 011' => return Sys TLBI;     // VAAE11S
    when '000 1000 0011 101' => return Sys TLBI;     // VALE11S
    when '000 1000 0011 111' => return Sys TLBI;     // VAALE11S
    when '000 1000 0101 001' => return Sys TLBI;     // RVAE10S
    when '000 1000 0101 011' => return Sys TLBI;     // RVAAE10S
    when '000 1000 0101 101' => return Sys TLBI;     // RVALE10S
    when '000 1000 0101 111' => return Sys TLBI;     // RVAALE10S
    when '000 1000 0110 001' => return Sys TLBI;     // RVAE1
    when '000 1000 0110 011' => return Sys TLBI;     // RVAAE1
    when '000 1000 0110 101' => return Sys TLBI;     // RVALE1
    when '000 1000 0110 111' => return Sys TLBI;     // RVAALE1
    when '000 1000 0111 000' => return Sys TLBI;     // VMALLE1
    when '000 1000 0111 001' => return Sys TLBI;     // VAE1
    when '000 1000 0111 010' => return Sys TLBI;     // ASIDE1
  end case
end

```

```

when '000 1000 0111 011' => return Sys TLBI; // VAAE1
when '000 1000 0111 101' => return Sys TLBI; // VALE1
when '000 1000 0111 111' => return Sys TLBI; // VAALE1
when '000 1001 0001 000' => return Sys TLBI; // VMALLE1OSNXS
when '000 1001 0001 001' => return Sys TLBI; // VAE1OSNXS
when '000 1001 0001 010' => return Sys TLBI; // ASIDE1OSNXS
when '000 1001 0001 011' => return Sys TLBI; // VAAE1OSNXS
when '000 1001 0001 101' => return Sys TLBI; // VALE1OSNXS
when '000 1001 0001 111' => return Sys TLBI; // VAALE1OSNXS
when '000 1001 0010 001' => return Sys TLBI; // RVAE1ISNXS
when '000 1001 0010 011' => return Sys TLBI; // RVAAE1ISNXS
when '000 1001 0010 101' => return Sys TLBI; // RVALE1ISNXS
when '000 1001 0010 111' => return Sys TLBI; // RVAAE1ISNXS
when '000 1001 0011 000' => return Sys TLBI; // VMALLE1ISNXS
when '000 1001 0011 001' => return Sys TLBI; // VAE1ISNXS
when '000 1001 0011 010' => return Sys TLBI; // ASIDE1ISNXS
when '000 1001 0011 011' => return Sys TLBI; // VAAE1ISNXS
when '000 1001 0011 101' => return Sys TLBI; // VALE1ISNXS
when '000 1001 0011 111' => return Sys TLBI; // VAALE1ISNXS
when '000 1001 0101 001' => return Sys TLBI; // RVAE1OSNXS
when '000 1001 0101 011' => return Sys TLBI; // RVAAE1OSNXS
when '000 1001 0101 101' => return Sys TLBI; // RVALE1OSNXS
when '000 1001 0101 111' => return Sys TLBI; // RVAAE1OSNXS
when '000 1001 0110 001' => return Sys TLBI; // RVAE1NXS
when '000 1001 0110 011' => return Sys TLBI; // RVAAE1NXS
when '000 1001 0110 101' => return Sys TLBI; // RVALE1NXS
when '000 1001 0110 111' => return Sys TLBI; // RVAAE1NXS
when '000 1001 0111 000' => return Sys TLBI; // VMALLE1NXS
when '000 1001 0111 001' => return Sys TLBI; // VAE1NXS
when '000 1001 0111 010' => return Sys TLBI; // ASIDE1NXS
when '000 1001 0111 011' => return Sys TLBI; // VAAE1NXS
when '000 1001 0111 101' => return Sys TLBI; // VALE1NXS
when '000 1001 0111 111' => return Sys TLBI; // VAALE1NXS
when '100 1000 0000 001' => return Sys TLBI; // IPAS2E1IS
when '100 1000 0000 010' => return Sys TLBI; // RIPAS2E1IS
when '100 1000 0000 101' => return Sys TLBI; // IPAS2LE1IS
when '100 1000 0000 110' => return Sys TLBI; // RIPAS2LE1IS
when '100 1000 0001 000' => return Sys TLBI; // ALLE2OS
when '100 1000 0001 001' => return Sys TLBI; // VAE2OS
when '100 1000 0001 100' => return Sys TLBI; // ALLE1OS
when '100 1000 0001 101' => return Sys TLBI; // VALE2OS
when '100 1000 0001 110' => return Sys TLBI; // VMALLS12E1OS
when '100 1000 0010 001' => return Sys TLBI; // RVAE2IS
when '100 1000 0010 101' => return Sys TLBI; // RVALE2IS
when '100 1000 0011 000' => return Sys TLBI; // ALLE2IS
when '100 1000 0011 001' => return Sys TLBI; // VAE2IS
when '100 1000 0011 100' => return Sys TLBI; // ALLE1IS
when '100 1000 0011 101' => return Sys TLBI; // VALE2IS
when '100 1000 0011 110' => return Sys TLBI; // VMALLS12E1IS
when '100 1000 0100 000' => return Sys TLBI; // IPAS2E1OS
when '100 1000 0100 001' => return Sys TLBI; // IPAS2E1
when '100 1000 0100 010' => return Sys TLBI; // RIPAS2E1
when '100 1000 0100 011' => return Sys TLBI; // RIPAS2E1OS
when '100 1000 0100 100' => return Sys TLBI; // IPAS2LE1OS
when '100 1000 0100 101' => return Sys TLBI; // IPAS2LE1
when '100 1000 0100 110' => return Sys TLBI; // RIPAS2LE1
when '100 1000 0100 111' => return Sys TLBI; // RIPAS2LE1OS
when '100 1000 0101 001' => return Sys TLBI; // RVAE2OS
when '100 1000 0101 101' => return Sys TLBI; // RVALE2OS
when '100 1000 0110 001' => return Sys TLBI; // RVAE2
when '100 1000 0110 101' => return Sys TLBI; // RVALE2
when '100 1000 0111 000' => return Sys TLBI; // ALLE2
when '100 1000 0111 001' => return Sys TLBI; // VAE2
when '100 1000 0111 100' => return Sys TLBI; // ALLE1
when '100 1000 0111 101' => return Sys TLBI; // VALE2
when '100 1000 0111 110' => return Sys TLBI; // VMALLS12E1
when '100 1001 0000 001' => return Sys TLBI; // IPAS2E1ISNXS
when '100 1001 0000 010' => return Sys TLBI; // RIPAS2E1ISNXS
when '100 1001 0000 101' => return Sys TLBI; // IPAS2LE1ISNXS
when '100 1001 0000 110' => return Sys TLBI; // RIPAS2LE1ISNXS
when '100 1001 0001 000' => return Sys TLBI; // ALLE2OSNXS
when '100 1001 0001 001' => return Sys TLBI; // VAE2OSNXS
when '100 1001 0001 100' => return Sys TLBI; // ALLE1OSNXS
when '100 1001 0001 101' => return Sys TLBI; // VALE2OSNXS
when '100 1001 0001 110' => return Sys TLBI; // VMALLS12E1OSNXS
when '100 1001 0010 001' => return Sys TLBI; // RVAE2ISNXS
when '100 1001 0010 101' => return Sys TLBI; // RVALE2ISNXS
when '100 1001 0011 000' => return Sys TLBI; // ALLE2ISNXS
when '100 1001 0011 001' => return Sys TLBI; // VAE2ISNXS
when '100 1001 0011 100' => return Sys TLBI; // ALLE1ISNXS
when '100 1001 0011 101' => return Sys TLBI; // VALE2ISNXS
when '100 1001 0011 110' => return Sys TLBI; // VMALLS12E1ISNXS
when '100 1001 0100 000' => return Sys TLBI; // IPAS2E1OSNXS
when '100 1001 0100 001' => return Sys TLBI; // IPAS2E1NXS
when '100 1001 0100 010' => return Sys TLBI; // RIPAS2E1NXS
when '100 1001 0100 011' => return Sys TLBI; // RIPAS2E1OSNXS
when '100 1001 0100 100' => return Sys TLBI; // IPAS2LE1OSNXS
when '100 1001 0100 101' => return Sys TLBI; // IPAS2LE1NXS
when '100 1001 0100 110' => return Sys TLBI; // RIPAS2LE1NXS

```

```

when '100 1001 0100 111' => return Sys_TLBI; // RIPAS2LE1OSNXS
when '100 1001 0101 001' => return Sys_TLBI; // RVAE2OSNXS
when '100 1001 0101 101' => return Sys_TLBI; // RVALE2OSNXS
when '100 1001 0110 001' => return Sys_TLBI; // RVAE2NXS
when '100 1001 0110 101' => return Sys_TLBI; // RVALE2NXS
when '100 1001 0111 000' => return Sys_TLBI; // ALLE2NXS
when '100 1001 0111 001' => return Sys_TLBI; // VAE2NXS
when '100 1001 0111 100' => return Sys_TLBI; // ALLE1NXS
when '100 1001 0111 101' => return Sys_TLBI; // VALE2NXS
when '100 1001 0111 110' => return Sys_TLBI; // VMALLS12E1NXS
when '110 1000 0001 000' => return Sys_TLBI; // ALLE3OS
when '110 1000 0001 001' => return Sys_TLBI; // VAE3OS
when '110 1000 0001 100' => return Sys_TLBI; // PAALLOS
when '110 1000 0001 101' => return Sys_TLBI; // VALE3OS
when '110 1000 0010 001' => return Sys_TLBI; // RVAE3IS
when '110 1000 0010 101' => return Sys_TLBI; // RVALE3IS
when '110 1000 0011 000' => return Sys_TLBI; // ALLE3IS
when '110 1000 0011 001' => return Sys_TLBI; // VAE3IS
when '110 1000 0011 101' => return Sys_TLBI; // VALE3IS
when '110 1000 0100 011' => return Sys_TLBI; // RPAOS
when '110 1000 0100 111' => return Sys_TLBI; // RPALOS
when '110 1000 0101 001' => return Sys_TLBI; // RVAE3OS
when '110 1000 0101 101' => return Sys_TLBI; // RVALE3OS
when '110 1000 0110 001' => return Sys_TLBI; // RVAE3
when '110 1000 0110 101' => return Sys_TLBI; // RVALE3
when '110 1000 0111 000' => return Sys_TLBI; // ALLE3
when '110 1000 0111 001' => return Sys_TLBI; // VAE3
when '110 1000 0111 100' => return Sys_TLBI; // PAALL
when '110 1000 0111 101' => return Sys_TLBI; // VALE3
when '110 1001 0001 000' => return Sys_TLBI; // ALLE3OSNXS
when '110 1001 0001 001' => return Sys_TLBI; // VAE3OSNXS
when '110 1001 0001 101' => return Sys_TLBI; // VALE3OSNXS
when '110 1001 0010 001' => return Sys_TLBI; // RVAE3ISNXS
when '110 1001 0010 101' => return Sys_TLBI; // RVALE3ISNXS
when '110 1001 0011 000' => return Sys_TLBI; // ALLE3ISNXS
when '110 1001 0011 001' => return Sys_TLBI; // VAE3ISNXS
when '110 1001 0011 101' => return Sys_TLBI; // VALE3ISNXS
when '110 1001 0101 001' => return Sys_TLBI; // RVAE3OSNXS
when '110 1001 0101 101' => return Sys_TLBI; // RVALE3OSNXS
when '110 1001 0110 001' => return Sys_TLBI; // RVAE3NXS
when '110 1001 0110 101' => return Sys_TLBI; // RVALE3NXS
when '110 1001 0111 000' => return Sys_TLBI; // ALLE3NXS
when '110 1001 0111 001' => return Sys_TLBI; // VAE3NXS
when '110 1001 0111 101' => return Sys_TLBI; // VALE3NXS
otherwise => return Sys_SYS;
end;
end;

```

Library pseudocode for aarch64/functions/sysop/SystemOp

```

// SystemOp
// =====
// System instruction types.

type SystemOp of enumeration {
    Sys_AT,
    Sys_BRB,
    Sys_DC,
    Sys_IC,
    Sys_TLBI,
    Sys_SYS
};

```



```
// SysOp128()
// =====

func SysOp128(op1 : bits(3), CRn : bits(4), CRm : bits(4), op2 : bits(3)) => SystemOp128
begin
  case op1::CRn::CRm::op2 of
    when '000 1000 0001 001' => return Sys TLBIP; // VAE10S
    when '000 1000 0001 011' => return Sys TLBIP; // VAAE10S
    when '000 1000 0001 101' => return Sys TLBIP; // VALE10S
    when '000 1000 0001 111' => return Sys TLBIP; // VAALE10S
    when '000 1000 0011 001' => return Sys TLBIP; // VAE1IS
    when '000 1000 0011 011' => return Sys TLBIP; // VAAE1IS
    when '000 1000 0011 101' => return Sys TLBIP; // VALE1IS
    when '000 1000 0011 111' => return Sys TLBIP; // VAALE1IS
    when '000 1000 0111 001' => return Sys TLBIP; // VAE1
    when '000 1000 0111 011' => return Sys TLBIP; // VAAE1
    when '000 1000 0111 101' => return Sys TLBIP; // VALE1
    when '000 1000 0111 111' => return Sys TLBIP; // VAALE1
    when '000 1001 0001 001' => return Sys TLBIP; // VAE10SNXS
    when '000 1001 0001 011' => return Sys TLBIP; // VAAE10SNXS
    when '000 1001 0001 101' => return Sys TLBIP; // VALE10SNXS
    when '000 1001 0001 111' => return Sys TLBIP; // VAALE10SNXS
    when '000 1001 0011 001' => return Sys TLBIP; // VAE1ISNXS
    when '000 1001 0011 011' => return Sys TLBIP; // VAAE1ISNXS
    when '000 1001 0011 101' => return Sys TLBIP; // VALE1ISNXS
    when '000 1001 0011 111' => return Sys TLBIP; // VAALE1ISNXS
    when '000 1001 0111 001' => return Sys TLBIP; // VAE1NXS
    when '000 1001 0111 011' => return Sys TLBIP; // VAAE1NXS
    when '000 1001 0111 101' => return Sys TLBIP; // VALE1NXS
    when '000 1001 0111 111' => return Sys TLBIP; // VAALE1NXS
    when '100 1000 0001 001' => return Sys TLBIP; // VAE20S
    when '100 1000 0001 101' => return Sys TLBIP; // VALE20S
    when '100 1000 0011 001' => return Sys TLBIP; // VAE2IS
    when '100 1000 0011 101' => return Sys TLBIP; // VALE2IS
    when '100 1000 0111 001' => return Sys TLBIP; // VAE2
    when '100 1000 0111 101' => return Sys TLBIP; // VALE2
    when '100 1001 0001 001' => return Sys TLBIP; // VAE20SNXS
    when '100 1001 0001 101' => return Sys TLBIP; // VALE20SNXS
    when '100 1001 0011 001' => return Sys TLBIP; // VAE2ISNXS
    when '100 1001 0011 101' => return Sys TLBIP; // VALE2ISNXS
    when '100 1001 0111 001' => return Sys TLBIP; // VAE2NXS
    when '100 1001 0111 101' => return Sys TLBIP; // VALE2NXS
    when '110 1000 0001 001' => return Sys TLBIP; // VAE30S
    when '110 1000 0001 101' => return Sys TLBIP; // VALE30S
    when '110 1000 0011 001' => return Sys TLBIP; // VAE3IS
    when '110 1000 0011 101' => return Sys TLBIP; // VALE3IS
    when '110 1000 0111 001' => return Sys TLBIP; // VAE3
    when '110 1000 0111 101' => return Sys TLBIP; // VALE3
    when '110 1001 0001 001' => return Sys TLBIP; // VAE30SNXS
    when '110 1001 0001 101' => return Sys TLBIP; // VALE30SNXS
    when '110 1001 0011 001' => return Sys TLBIP; // VAE3ISNXS
    when '110 1001 0011 101' => return Sys TLBIP; // VALE3ISNXS
    when '110 1001 0111 001' => return Sys TLBIP; // VAE3NXS
    when '110 1001 0111 101' => return Sys TLBIP; // VALE3NXS
    when '100 1000 0000 001' => return Sys TLBIP; // IPAS2E1IS
    when '100 1000 0000 101' => return Sys TLBIP; // IPAS2LE1IS
    when '100 1000 0100 000' => return Sys TLBIP; // IPAS2E10S
    when '100 1000 0100 001' => return Sys TLBIP; // IPAS2E1
    when '100 1000 0100 100' => return Sys TLBIP; // IPAS2LE10S
    when '100 1000 0100 101' => return Sys TLBIP; // IPAS2LE1
    when '100 1001 0000 001' => return Sys TLBIP; // IPAS2E1ISNXS
    when '100 1001 0000 101' => return Sys TLBIP; // IPAS2LE1ISNXS
    when '100 1001 0100 000' => return Sys TLBIP; // IPAS2E10SNXS
    when '100 1001 0100 001' => return Sys TLBIP; // IPAS2E1NXS
    when '100 1001 0100 100' => return Sys TLBIP; // IPAS2LE10SNXS
    when '100 1001 0100 101' => return Sys TLBIP; // IPAS2LE1NXS
    when '000 1000 0010 001' => return Sys TLBIP; // RVAE1IS
    when '000 1000 0010 011' => return Sys TLBIP; // RVAE1IS
    when '000 1000 0010 101' => return Sys TLBIP; // RVALE1IS
    when '000 1000 0010 111' => return Sys TLBIP; // RVALE1IS
    when '000 1000 0101 001' => return Sys TLBIP; // RVAE10S
    when '000 1000 0101 011' => return Sys TLBIP; // RVAE10S
    when '000 1000 0101 101' => return Sys TLBIP; // RVALE10S
    when '000 1000 0101 111' => return Sys TLBIP; // RVALE10S
    when '000 1000 0110 001' => return Sys TLBIP; // RVAE1
    when '000 1000 0110 011' => return Sys TLBIP; // RVAE1
    when '000 1000 0110 101' => return Sys TLBIP; // RVALE1
    when '000 1000 0110 111' => return Sys TLBIP; // RVALE1
    when '000 1001 0010 001' => return Sys TLBIP; // RVAE1ISNXS
    when '000 1001 0010 011' => return Sys TLBIP; // RVAE1ISNXS
    when '000 1001 0010 101' => return Sys TLBIP; // RVALE1ISNXS
    when '000 1001 0010 111' => return Sys TLBIP; // RVALE1ISNXS
    when '000 1001 0101 001' => return Sys TLBIP; // RVAE10SNXS
    when '000 1001 0101 011' => return Sys TLBIP; // RVAE10SNXS
    when '000 1001 0101 101' => return Sys TLBIP; // RVALE10SNXS
    when '000 1001 0101 111' => return Sys TLBIP; // RVALE10SNXS
    when '000 1001 0110 001' => return Sys TLBIP; // RVAE1NXS
    when '000 1001 0110 011' => return Sys TLBIP; // RVAE1NXS
    when '000 1001 0110 101' => return Sys TLBIP; // RVALE1NXS
```

```

when '000 1001 0110 111' => return Sys_TLBIP; // RVAAL1NXS
when '100 1000 0010 001' => return Sys_TLBIP; // RVAE2IS
when '100 1000 0010 101' => return Sys_TLBIP; // RVALE2IS
when '100 1000 0101 001' => return Sys_TLBIP; // RVAE2OS
when '100 1000 0101 101' => return Sys_TLBIP; // RVALE2OS
when '100 1000 0110 001' => return Sys_TLBIP; // RVAE2
when '100 1000 0110 101' => return Sys_TLBIP; // RVALE2
when '100 1001 0010 001' => return Sys_TLBIP; // RVAE2ISNXS
when '100 1001 0010 101' => return Sys_TLBIP; // RVALE2ISNXS
when '100 1001 0101 001' => return Sys_TLBIP; // RVAE2OSNXS
when '100 1001 0101 101' => return Sys_TLBIP; // RVALE2OSNXS
when '100 1001 0110 001' => return Sys_TLBIP; // RVAE2NXS
when '100 1001 0110 101' => return Sys_TLBIP; // RVALE2NXS
when '110 1000 0010 001' => return Sys_TLBIP; // RVAE3IS
when '110 1000 0010 101' => return Sys_TLBIP; // RVALE3IS
when '110 1000 0101 001' => return Sys_TLBIP; // RVAE3OS
when '110 1000 0101 101' => return Sys_TLBIP; // RVALE3OS
when '110 1000 0110 001' => return Sys_TLBIP; // RVAE3
when '110 1000 0110 101' => return Sys_TLBIP; // RVALE3
when '110 1001 0010 001' => return Sys_TLBIP; // RVAE3ISNXS
when '110 1001 0010 101' => return Sys_TLBIP; // RVALE3ISNXS
when '110 1001 0101 001' => return Sys_TLBIP; // RVAE3OSNXS
when '110 1001 0101 101' => return Sys_TLBIP; // RVALE3OSNXS
when '110 1001 0110 001' => return Sys_TLBIP; // RVAE3NXS
when '110 1001 0110 101' => return Sys_TLBIP; // RVALE3NXS
when '100 1000 0000 010' => return Sys_TLBIP; // RIPAS2E1IS
when '100 1000 0000 110' => return Sys_TLBIP; // RIPAS2LE1IS
when '100 1000 0100 010' => return Sys_TLBIP; // RIPAS2E1
when '100 1000 0100 011' => return Sys_TLBIP; // RIPAS2E1OS
when '100 1000 0100 110' => return Sys_TLBIP; // RIPAS2LE1
when '100 1000 0100 111' => return Sys_TLBIP; // RIPAS2LE1OS
when '100 1001 0000 010' => return Sys_TLBIP; // RIPAS2E1ISNXS
when '100 1001 0000 110' => return Sys_TLBIP; // RIPAS2LE1ISNXS
when '100 1001 0100 010' => return Sys_TLBIP; // RIPAS2E1NXS
when '100 1001 0100 011' => return Sys_TLBIP; // RIPAS2E1OSNXS
when '100 1001 0100 110' => return Sys_TLBIP; // RIPAS2LE1NXS
when '100 1001 0100 111' => return Sys_TLBIP; // RIPAS2LE1OSNXS
otherwise => return Sys_SYSP;
end;
end;

```

Library pseudocode for aarch64/functions/sysop_128/SystemOp128

```

// SystemOp128()
// =====
// System instruction types.

type SystemOp128 of enumeration {Sys_TLBIP, Sys_SYSP};

```

Library pseudocode for aarch64/functions/sysregisters/ELR_EL

```

// ELR_EL - accessor
// =====

accessor ELR_EL(el : bits(2)) <=> value : bits(64)
begin
  getter
    var r : bits(64);
    case el of
      when EL1 => r = ELR_EL1();
      when EL2 => r = ELR_EL2();
      when EL3 => r = ELR_EL3();
      otherwise => unreachable;
    end;
    return r;
  end;
  setter
    let r : bits(64) = value;
    case el of
      when EL1 => ELR_EL1() = r;
      when EL2 => ELR_EL2() = r;
      when EL3 => ELR_EL3() = r;
      otherwise => unreachable;
    end;
  end;
end;
end;

```

Library pseudocode for aarch64/functions/sysregisters/ELR_ELx

```
// ELR_ELx - accessor
// =====

accessor ELR_ELx() <=> value : bits(64)
begin
  getter
    assert PSTATE.EL != EL0;
    return ELR\_EL(PSTATE.EL);
  end;
  setter
    assert PSTATE.EL != EL0;
    ELR\_EL(PSTATE.EL) = value;
  end;
end;
```

Library pseudocode for aarch64/functions/sysregisters/ESR_EL

```
// ESR_EL - accessor
// =====

accessor ESR_EL(regime : bits(2)) <=> value : ESRTYPE
begin
  getter
    var r : bits(64);
    case regime of
      when EL1 => r = ESR_EL1();
      when EL2 => r = ESR_EL2();
      when EL3 => r = ESR_EL3();
      otherwise => unreachable;
    end;
    return r;
  end;
  setter
    let r : bits(64) = value;
    case regime of
      when EL1 => ESR_EL1() = r;
      when EL2 => ESR_EL2() = r;
      when EL3 => ESR_EL3() = r;
      otherwise => unreachable;
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/sysregisters/ESR_ELx

```
// ESR_ELx - accessor
// =====

accessor ESR_ELx() <=> value : ESRTYPE
begin
  getter
    return ESR\_EL(S1TranslationRegime());
  end;
  setter
    ESR\_EL(S1TranslationRegime()) = value;
  end;
end;
```

Library pseudocode for aarch64/functions/sysregisters/FAR_EL

```
// FAR_EL - accessor
// =====

accessor FAR_EL(regime : bits(2)) <=> value : bits(64)
begin
  getter
    var r : bits(64);
    case regime of
      when EL1 => r = FAR_EL1();
      when EL2 => r = FAR_EL2();
      when EL3 => r = FAR_EL3();
      otherwise => unreachable;
    end;
    return r;
  end;
  setter
    let r : bits(64) = value;
    case regime of
      when EL1 => FAR_EL1() = r;
      when EL2 => FAR_EL2() = r;
      when EL3 => FAR_EL3() = r;
      otherwise => unreachable;
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/sysregisters/FAR_ELx

```
// FAR_ELx - accessor
// =====

accessor FAR_ELx() <=> value : bits(64)
begin
  getter
    return FAR_EL(S1TranslationRegime());
  end;
  setter
    FAR_EL(S1TranslationRegime()) = value;
  end;
end;
```

Library pseudocode for aarch64/functions/sysregisters/PFAR_EL

```
// PFAR_EL - accessor
// =====

accessor PFAR_EL(regime : bits(2)) <=> value : bits(64)
begin
  getter
    assert (IsFeatureImplemented(FEAT_P FAR) ||
      (regime == EL3 && IsFeatureImplemented(FEAT_RME)));
    var r : bits(64);
    case regime of
      when EL1 => r = PFAR_EL1();
      when EL2 => r = PFAR_EL2();
      when EL3 => r = MFAR_EL3();
      otherwise => unreachable;
    end;
    return r;
  end;
  setter
    let r : bits(64) = value;
    assert (IsFeatureImplemented(FEAT_P FAR) ||
      (IsFeatureImplemented(FEAT_RME) && regime == EL3));
    case regime of
      when EL1 => PFAR_EL1() = r;
      when EL2 => PFAR_EL2() = r;
      when EL3 => MFAR_EL3() = r;
      otherwise => unreachable;
    end;
  end;
end;
```


Library pseudocode for aarch64/functions/sysregisters/PFAR_ELx

```
// PFAR_ELx - accessor
// =====

accessor PFAR_ELx() <=> value : bits(64)
begin
  getter
    return PFAR\_EL\(S1TranslationRegime\(\)\);
  end;

  setter
    PFAR\_EL\(S1TranslationRegime\(\)\) = value;
  end;
end;
```

Library pseudocode for aarch64/functions/sysregisters/SCTLR_EL

```
// SCTLR_EL - accessor
// =====

accessor SCTLR_EL(regime : bits(2)) <=> value : SCTLRType
begin
  getter
    var r : bits(64);
    case regime of
      when EL1 => r = SCTLR_EL1();
      when EL2 => r = SCTLR_EL2();
      when EL3 => r = SCTLR_EL3();
      otherwise => unreachable;
    end;
    return r;
  end;
  setter
    let r : bits(64) = value;
    case regime of
      when EL1 => SCTLR_EL1() = r;
      when EL2 => SCTLR_EL2() = r;
      when EL3 => SCTLR_EL3() = r;
      otherwise => unreachable;
    end;
  end;
end;
```

Library pseudocode for aarch64/functions/sysregisters/SCTLR_ELx

```
// SCTLR_ELx - accessor
// =====

accessor SCTLR_ELx() <=> value : SCTLRType
begin
  getter
    return SCTLR\_EL\(S1TranslationRegime\(\)\);
  end;
  setter
    SCTLR\_EL\(S1TranslationRegime\(\)\) = value;
  end;
end;
```

Library pseudocode for aarch64/functions/sysregisters/VBAR_EL

```
// VBAR_EL - accessor
// =====

accessor VBAR_EL(regime : bits(2)) <=> value : bits(64)
begin
  getter
    var r : bits(64);
    case regime of
      when EL1 => r = VBAR_EL1();
      when EL2 => r = VBAR_EL2();
      when EL3 => r = VBAR_EL3();
      otherwise => unreachable;
    end;
    return r;
  end;
  setter
    pass;
  end;
end;
```

Library pseudocode for aarch64/functions/sysregisters/VBAR_ELx

```
// VBAR_ELx - accessor
// =====

accessor VBAR_ELx() <=> value : bits(64)
begin
    getter
        return VBAR\_EL\(S1TranslationRegime\(\)\);
    end;
    setter
        VBAR\_EL\(S1TranslationRegime\(\)\)[63:0] = value;
    end;
end;
```

Library pseudocode for aarch64/functions/system/AArch64_CheckDAIFAccess

```
// AArch64_CheckDAIFAccess()
// =====
// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted.

func AArch64_CheckDAIFAccess(field : PSTATEField)
begin
    if PSTATE.EL == EL0 && field IN {PSTATEField DAIFSet, PSTATEField DAIFClr} then
        if IsInHost\(\) || SCTLR\_EL1\(\).UMA == '0' then
            if EL2Enabled\(\) && HCR\_EL2\(\).TGE == '1' then
                AArch64\_SystemAccessTrap\(EL2, 0x18\);
            else
                AArch64\_SystemAccessTrap\(EL1, 0x18\);
            end;
        end;
    end;
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ChooseNonExcludedTag

```
// AArch64_ChooseNonExcludedTag()
// =====
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

func AArch64_ChooseNonExcludedTag(tag_in : bits(4), offset_in : bits(4),
                                   exclude : bits(16)) => bits(4)
begin
    var tag : bits(4) = tag_in;
    var offset : bits(4) = offset_in;

    if IsOnes(exclude) then
        return '0000';
    end;

    if offset == '0000' then
        while exclude[UInt(tag)] == '1' looplimit 16 do
            tag = tag + '0001';
        end;
    end;

    while offset != '0000' looplimit 16 do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude[UInt(tag)] == '1' looplimit 16 do
            tag = tag + '0001';
        end;
    end;

    return tag;
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ChooseNonExcludedTagOrZero

```
// AArch64_ChooseNonExcludedTagOrZero()
// =====
// Return a tag derived from the start and the offset values, excluding any
// tags in the given mask, or zero if Allocation Tag access is not enabled.

func AArch64_ChooseNonExcludedTagOrZero(tag : bits(4), offset : bits(4),
                                          exclude : bits(16)) => bits(4)
begin
    if IsMTEEnabled\(PSTATE.EL\) then
        return AArch64\_ChooseNonExcludedTag(tag, offset, exclude);
    else
        return '0000';
    end;
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ChooseTagOrZero

```
// AArch64_ChooseTagOrZero()
// =====
// Return a tag, excluding any tags in the given mask, , or zero if Allocation
// Tag access is not enabled.

func AArch64_ChooseTagOrZero(exclude : bits(16)) => bits(4)
begin
    if IsMTEEnabled(PSTATE.EL) then
        if GCR_EL1().RRND == '1' then
            if IsOnes(exclude) then
                return '0000';
            else
                return ChooseRandomNonExcludedTag(exclude);
            end;
        else
            let start_tag : bits(4) = RGSR_EL1().TAG;
            let offset : bits(4) = AArch64_RandomTag();

            RGSR_EL1().TAG = AArch64_ChooseNonExcludedTag(start_tag, offset, exclude);

            return RGSR_EL1().TAG;
        end;
    else
        return '0000';
    end;
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ExecutingERETInstr

```
// AArch64_ExecutingERETInstr()
// =====
// Returns TRUE if current instruction is ERET.

func AArch64_ExecutingERETInstr() => boolean
begin
    let instr : bits(32) = ThisInstr();
    return instr[31:12] == '11010110100111110000';
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ImpDefSysInstr

```
// AArch64_ImpDefSysInstr()
// =====
// Execute an implementation-defined system instruction with write (source operand).

impdef func AArch64_ImpDefSysInstr(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                   crm : bits(4), op2 : bits(3), t : integer)
begin
    Undefined();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ImpDefSysInstr128

```
// AArch64_ImpDefSysInstr128()
// =====
// Execute an implementation-defined system instruction with write (128-bit source operand).

impdef func AArch64_ImpDefSysInstr128(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                       crm : bits(4), op2 : bits(3),
                                       t : integer, t2 : integer)
begin
    Undefined();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ImpDefSysInstrWithResult

```
// AArch64_ImpDefSysInstrWithResult()
// =====
// Execute an implementation-defined system instruction with read (result operand).

impdef func AArch64_ImpDefSysInstrWithResult(op0 : bits(2),
                                              op1 : bits(3),
                                              crn : bits(4),
                                              crm : bits(4),
                                              op2 : bits(3),
                                              t : integer)
begin
    Undefined();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ImpDefSysRegRead

```
// AArch64_ImpDefSysRegRead()
// =====
// Read from an implementation-defined System register and write the contents of the register
// to X[t].

impdef func AArch64_ImpDefSysRegRead(op0 : bits(2), op1 : bits(3),
                                     crn : bits(4), crm : bits(4),
                                     op2 : bits(3), t : integer)

begin
    Undefined();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ImpDefSysRegRead128

```
// AArch64_ImpDefSysRegRead128()
// =====
// Read from an 128-bit implementation-defined System register
// and write the contents of the register to X[t], X[t+1].

impdef func AArch64_ImpDefSysRegRead128(op0 : bits(2),
                                         op1 : bits(3), crn : bits(4),
                                         crm : bits(4), op2 : bits(3),
                                         t : integer, t2 : integer)

begin
    Undefined();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ImpDefSysRegWrite

```
// AArch64_ImpDefSysRegWrite()
// =====
// Write to an implementation-defined System register.

impdef func AArch64_ImpDefSysRegWrite(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                       crm : bits(4), op2 : bits(3), t : integer)

begin
    Undefined();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_ImpDefSysRegWrite128

```
// AArch64_ImpDefSysRegWrite128()
// =====
// Write the contents of X[t], X[t+1] to an 128-bit implementation-defined System register.

impdef func AArch64_ImpDefSysRegWrite128(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                          crm : bits(4), op2 : bits(3),
                                          t : integer, t2 : integer)

begin
    Undefined();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_InterruptPending

```
// AArch64_InterruptPending()
// =====
// Returns TRUE if there are any pending physical, virtual, or delegated
// interrupts, and FALSE otherwise.

func AArch64_InterruptPending() => boolean
begin
    let (irq_pending, -) = IRQPending();
    let (fiq_pending, -) = FIQPending();
    let pending_physical_interrupt : boolean = (irq_pending || fiq_pending ||
                                                IsPhysicalSErrorPending());

    var pending_virtual_interrupt : boolean = FALSE;
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2().TGE == '0' then
        let virq_pending : boolean = (HCR_EL2().IMO == '1' && (VirtualIRQPending() ||
                                                                HCR_EL2().VI == '1'));
        let vfiq_pending : boolean = (HCR_EL2().FMO == '1' && (VirtualFIQPending() ||
                                                                HCR_EL2().VF == '1'));
        let vsei_pending : boolean = ((HCR_EL2().AMO == '1' ||
                                        (IsFeatureImplemented(FEAT_DoubleFault2) &&
                                         IsHCRXEL2Enabled() && HCRX_EL2().TMEA == '1')) &&
                                       (IsVirtualSErrorPending() || HCR_EL2().VSE == '1'));

        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;
    end;

    let pending_delegated_interrupt : boolean = (PSTATE.EL != EL3 &&
                                                EffectiveSCR_EL3_EndSE() == '1' &&
                                                SCR_EL3().DSE == '1');

    return pending_physical_interrupt || pending_virtual_interrupt || pending_delegated_interrupt;
end;
```

Library pseudocode for aarch64/functions/system/AArch64_NextRandomTagBit

```
// AArch64_NextRandomTagBit()
// =====
// Generate a random bit suitable for generating a random Allocation Tag.

func AArch64_NextRandomTagBit() => bit
begin
    assert GCR_EL1().RRND == '0';
    let lfsr : bits(16) = RGSR_EL1().SEED[15:0];
    let top : bit = lfsr[5] XOR lfsr[3] XOR lfsr[2] XOR lfsr[0];
    RGSR_EL1().SEED[15:0] = top::lfsr[15:1];
    return top;
end;
```

Library pseudocode for aarch64/functions/system/AArch64_RandomTag

```
// AArch64_RandomTag()
// =====
// Generate a random Allocation Tag.

func AArch64_RandomTag() => bits(4)
begin
    var tag : bits(4);
    for i = 0 to 3 do
        tag[i] = AArch64_NextRandomTagBit();
    end;
    return tag;
end;
```

Library pseudocode for aarch64/functions/system/AArch64_SysInstr

```
// AArch64_SysInstr()
// =====
// Execute a system instruction with write (source operand).

impdef func AArch64_SysInstr(op0 : bits(2), op1 : bits(3),
                             crn : bits(4), crm : bits(4), op2 : bits(3), t : integer)
begin
    Unimplemented();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_SysInstrWithResult

```
// AArch64_SysInstrWithResult()
// =====
// Execute a system instruction with read (result operand).
// Writes the result of the instruction to X[t].

impdef func AArch64_SysInstrWithResult(op0 : bits(2), op1 : bits(3),
                                       crn : bits(4), crm : bits(4),
                                       op2 : bits(3), t : integer)

begin
    Unimplemented();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_SysRegRead

```
// AArch64_SysRegRead()
// =====
// Read from a System register and write the contents of the register to X[t].

impdef func AArch64_SysRegRead(op0 : bits(2), op1 : bits(3), crn : bits(4),
                               crm : bits(4), op2 : bits(3), t : integer)

begin
    Unimplemented();
end;
```

Library pseudocode for aarch64/functions/system/AArch64_SysRegWrite

```
// AArch64_SysRegWrite()
// =====
// Write to a System register.

impdef func AArch64_SysRegWrite(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                crm : bits(4), op2 : bits(3), t : integer)

begin
    Unimplemented();
end;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible

```
// BTypeCompatible
// =====
// Records the branch target compatibility.
// Returns TRUE if the branch target is compatible with PSTATE.BTYPE, else FALSE.

var BTypeCompatible : boolean;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible_BTI

```
// BTypeCompatible_BTI
// =====
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

func BTypeCompatible_BTI(hintcode : bits(2)) => boolean
begin
    case hintcode of
        when '00' =>
            return PSTATE.BTYPE == '00';
        when '01' =>
            return PSTATE.BTYPE != '11';
        when '10' =>
            return PSTATE.BTYPE != '10';
        when '11' =>
            return TRUE;
    end;
end;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible_PAC

```
// BTypeCompatible_PAC()
// =====
// Returns TRUE if PACIxSP or PACIxSPPC instructions are implicitly compatible with PSTATE.BTYPE,
// FALSE otherwise.

func BTypeCompatible_PAC(pacinst : PACInstType) => boolean
begin
    if PSTATE.BTYPE != '11' then
        return TRUE;
    else
        let index : integer = if PSTATE.EL == EL0 then 35 else 36;
        return SCTLRL_ELx()[index] == '0';
    end;
end;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible_PACIXSP

```
// BTypeCompatible_PACIXSP()
// =====
// Returns TRUE if PACIASP or PACIBSP instructions are implicitly compatible with PSTATE.BTYPE,
// FALSE otherwise.

func BTypeCompatible_PACIXSP() => boolean
begin
    let pacinst : PACInstType = PACIxSP;
    return BTypeCompatible_PAC(pacinst);
end;
```

Library pseudocode for aarch64/functions/system/BTypeNext

```
// BTypeNext
// =====
// Updated every cycle with a value that depends upon the instruction being executed. Assigned to
// PSTATE.BTYPE at the end of each cycle and then cleared to zero. Allows SPSR save/restore of
// BTYPE for the current instruction being executed.

var BTypeNext : bits(2);
```

Library pseudocode for aarch64/functions/system/ChooseRandomNonExcludedTag

```
// ChooseRandomNonExcludedTag()
// =====
// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
//
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation should select a tag from a uniform distribution and
// avoid common pitfalls such as modulo bias.
//
// This function can read RGSRL_EL1 and/or write RGSRL_EL1 to an IMPLEMENTATION DEFINED value.
// If it is not capable of writing RGSRL_EL1.SEED[15:0] to zero from a previous nonzero
// RGSRL_EL1.SEED value, it is IMPLEMENTATION DEFINED whether the randomness is significantly
// impacted if RGSRL_EL1.SEED[15:0] is set to zero.

impdef func ChooseRandomNonExcludedTag(exclude_in : bits(16)) => bits(4)
begin
    return ARBITRARY : bits(4);
end;
```

Library pseudocode for aarch64/functions/system/EffectiveBADDR

```
// EffectiveBADDR()
// =====
// Check if the given VA, held in a register BADDR field, is RESS and apply
// CONSTRAINED UNPREDICTABLE behavior if it is not.

func EffectiveBADDR(baddr_in : bits(64), directread : boolean) => bits(64)
begin
    var baddr : bits(64) = baddr_in;
    // Determine top bit position based on feature support.
    let n : integer{} = (if IsFeatureImplemented(FEAT_LVA3) then 57
                        else (if IsFeatureImplemented(FEAT_LVA) then 53
                            else 49));

    // Check if the upper bits of the base address form a valid sign extension.
    if baddr[63:n] != Replicate{64-n}(baddr[n-1]) then
        let c = ConstrainUnpredictable(Unpredictable_BADDR_RESS);
        assert c IN {Constraint_RESS, Constraint_ALLRESS, Constraint_FAULT};
        if c == Constraint_ALLRESS || (!directread && c == Constraint_RESS) then
            baddr[63:n] = Replicate{64-n}(baddr[n-1]);
        end;
        // If c == Constraint_FAULT, baddr will cause an Address Size fault when used.
    end;

    return baddr;
end;
```

Library pseudocode for aarch64/functions/system/EffectiveICC_SRE_EL2_Enable

```
// EffectiveICC_SRE_EL2_Enable()
// =====
// Returns the Effective value of ICC_SRE_EL2.Enable

func EffectiveICC_SRE_EL2_Enable() => bit
begin
    if (ImpDefBool("Treat ICC_SRE_EL2.SRE as RAO/WI") &&
        ImpDefBool("Treat ICC_SRE_EL2.Enable as RAO/WI")) then
        return '1';
    end;
    if ICC_SRE_EL2().SRE == '0' then
        return '1';
    end;

    return ICC_SRE_EL2().Enable;
end;
```

Library pseudocode for aarch64/functions/system/EffectiveICC_SRE_EL3_Enable

```
// EffectiveICC_SRE_EL3_Enable()
// =====
// Returns the Effective value of ICC_SRE_EL3.Enable

func EffectiveICC_SRE_EL3_Enable() => bit
begin
    return ICC_SRE_EL3().Enable;
end;
```

Library pseudocode for aarch64/functions/system/InGuardedPage

```
// InGuardedPage
// =====
// Records whether the currently fetched instruction was retrieved from a guarded page, will be
// TRUE if the GP bit in the page or block descriptor for the current instruction fetch was equal
// to one.

var InGuardedPage : boolean;
```


Library pseudocode for aarch64/functions/system/IsHCRXEL2Enabled

```
// IsHCRXEL2Enabled()
// =====
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

readonly func IsHCRXEL2Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_HCX) then return FALSE; end;
    if ELUsingAArch32(EL2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().HXEn == '0' then
        return FALSE;
    end;

    return EL2Enabled();
end;
```

Library pseudocode for aarch64/functions/system/IsMTEEnabled

```
// IsMTEEnabled()
// =====
// Returns TRUE if the currently selected MTE mechanism is enabled, and FALSE otherwise.

func IsMTEEnabled(el : bits(2)) => boolean
begin
    let regime : Regime = TranslationRegime(el);
    var selected : boolean = FALSE;

    if IsPMTESelected(el) then
        if HaveEL(EL3) && SCR_EL3().ATA == '0' && el IN {EL0, EL1, EL2} then
            return FALSE;
        end;

        if EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2().ATA == '0' && el IN {EL0, EL1} then
            return FALSE;
        end;

        selected = TRUE;
    end;

    if selected then
        case regime of
            when Regime_EL3 =>
                return SCTLR_EL3().ATA == '1';
            when Regime_EL2 =>
                return SCTLR_EL2().ATA == '1';
            when Regime_EL20 =>
                return if el == EL0 then SCTLR_EL2().ATA0 == '1' else SCTLR_EL2().ATA == '1';
            when Regime_EL10 =>
                return if el == EL0 then SCTLR_EL1().ATA0 == '1' else SCTLR_EL1().ATA == '1';
            otherwise => unreachable;
        end;
    end;

    return FALSE;
end;
```

Library pseudocode for aarch64/functions/system/IsPMTESelected

```
// IsPMTESelected()
// =====
// Returns TRUE if Physical MTE is selected for the translation regime
// associated with the EL, and FALSE otherwise

func IsPMTESelected(el : bits(2)) => boolean
begin
    if !IsFeatureImplemented(FEAT_MTE2) then
        return FALSE;
    end;

    return TRUE;
end;
```

Library pseudocode for aarch64/functions/system/IsSCTLR2EL1Enabled

```
// IsSCTLR2EL1Enabled()
// =====
// Returns TRUE if access to SCTLR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL1 returns 0 when access is not enabled.

func IsSCTLR2EL1Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_SCTLR2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        return FALSE;
    elsif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().SCTLR2En == '0')) then
        return FALSE;
    else
        return TRUE;
    end;
end;
```

Library pseudocode for aarch64/functions/system/IsSCTLR2EL2Enabled

```
// IsSCTLR2EL2Enabled()
// =====
// Returns TRUE if access to SCTLR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL2 returns 0 when access is not enabled.

func IsSCTLR2EL2Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_SCTLR2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().SCTLR2En == '0' then
        return FALSE;
    end;

    return EL2Enabled();
end;
```

Library pseudocode for aarch64/functions/system/IsSystemRegisterMaskingEnabled

```
// IsSystemRegisterMaskingEnabled()
// =====
// Returns TRUE if access to the MASK registers introduced by FEAT_SRMASK is enabled and the
// values in the register are not treated as 0.

func IsSystemRegisterMaskingEnabled(el : bits(2)) => boolean
begin
    assert IsFeatureImplemented(FEAT_SRMASK) && el IN {EL1, EL2};

    if HaveEL(EL3) && SCR_EL3().SRMASKEn == '0' then
        return FALSE;
    end;

    return el == EL2 || !EL2Enabled() || (IsHCRXEL2Enabled() && HCRX_EL2().SRMASKEn == '1');
end;
```

Library pseudocode for aarch64/functions/system/IsTCR2EL1Enabled

```
// IsTCR2EL1Enabled()
// =====
// Returns TRUE if access to TCR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL1 returns 0 when access is not enabled.

func IsTCR2EL1Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        return FALSE;
    elsif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2().TCR2En == '0')) then
        return FALSE;
    else
        return TRUE;
    end;
end;
```

Library pseudocode for aarch64/functions/system/IsTCR2EL2Enabled

```
// IsTCR2EL2Enabled()
// =====
// Returns TRUE if access to TCR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL2 returns 0 when access is not enabled.

func IsTCR2EL2Enabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE; end;
    if HaveEL(EL3) && SCR_EL3().TCR2En == '0' then
        return FALSE;
    end;

    return EL2Enabled();
end;
```

Library pseudocode for aarch64/functions/system/PACInstType

```
// PACInstType
// =====
type PACInstType of enumeration {
    PACIxSP,
    PACIxSPPC,
};
```

Library pseudocode for aarch64/functions/system/SetBTypeCompatible

```
// SetBTypeCompatible()
// =====
// Sets the value of BTypeCompatible global variable used by BTI

func SetBTypeCompatible(x : boolean)
begin
    BTypeCompatible = x;
end;
```

Library pseudocode for aarch64/functions/system/SetBTypeNext

```
// SetBTypeNext()
// =====
// Set the value of BTypeNext global variable used by BTI

func SetBTypeNext(x : bits(2))
begin
    BTypeNext = x;
end;
```

Library pseudocode for aarch64/functions/system/SetInGuardedPage

```
// SetInGuardedPage()
// =====
// Global state updated to denote if memory access is from a guarded page.

func SetInGuardedPage(guardedpage : boolean)
begin
    InGuardedPage = guardedpage;
end;
```

Library pseudocode for aarch64/functions/system128/AArch64_SysInstr128

```
// AArch64_SysInstr128()
// =====
// Execute a system instruction with write (2 64-bit source operands).

impdef func AArch64_SysInstr128(op0 : bits(2), op1 : bits(3), crn : bits(4),
                                crm : bits(4), op2 : bits(3), t : integer, t2 : integer)
begin
    Unimplemented();
end;
```

Library pseudocode for aarch64/functions/system128/AArch64_SysRegRead128

```
// AArch64_SysRegRead128()
// =====
// Read from a 128-bit System register and write the contents of the register to X[t] and X[t2].

impdef func AArch64_SysRegRead128(op0 : bits(2), op1 : bits(3),
                                   crn : bits(4), crm : bits(4),
                                   op2 : bits(3), t : integer, t2 : integer)

begin
    Unimplemented();
end;
```

Library pseudocode for aarch64/functions/system128/AArch64_SysRegWrite128

```
// AArch64_SysRegWrite128()
// =====
// Read the contents of X[t] and X[t2] and write the contents to a 128-bit System register.

impdef func AArch64_SysRegWrite128(op0 : bits(2), op1 : bits(3),
                                    crn : bits(4), crm : bits(4),
                                    op2 : bits(3), t : integer, t2 : integer)

begin
    Unimplemented();
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_EffectiveBroadcast

```
// AArch64_EffectiveBroadcast()
// =====

func AArch64_EffectiveBroadcast(broadcast_in : Broadcast) => Broadcast

begin
    var broadcast : Broadcast = broadcast_in;

    if PSTATE.EL != EL1 then return broadcast; end;

    if broadcast == Broadcast\_NSH && EL2Enabled() && HCR\_EL2\(\).FB == '1' then
        broadcast = Broadcast\_ForcedISH;
    end;

    return broadcast;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBIP_IPAS2

```
// AArch64_TLBIP_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

func AArch64_TLBIP_IPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, level : TLBILevel, attr : TLBIMemAttr,
                        Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = TRUE;
    r.level = level;
    r.attr = attr;
    r.ttl = Xt[47:44];
    r.address = ZeroExtend{64}(Xt[107:64] :: Zeros{12});
    r.d64 = r.ttl == '00xx';
    r.d128 = TRUE;

    case security of
        when SS_NonSecure =>
            r.ipaspace = PAS_NonSecure;
        when SS_Secure =>
            r.ipaspace = if Xt[63] == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm =>
            r.ipaspace = PAS_Realm;
        otherwise =>
            // Root security state does not have stage 2 translation
            unreachable;
    end;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBIP_RIPAS2

```
// AArch64_TLBIP_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// broadcast domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.

func AArch64_TLBIP_RIPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr : TLBIMemAttr,
    Xt : bits(128))

begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op = TLBIOp\_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = TRUE;
    r.level = level;
    r.attr = attr;
    r.ttl[1:0] = Xt[38:37];
    r.d64 = r.ttl[1:0] == '00';
    r.d128 = TRUE;

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return; end;

    case security of
        when SS\_NonSecure =>
            r.ipaspace = PAS\_NonSecure;
        when SS\_Secure =>
            r.ipaspace = if Xt[63] == '1' then PAS\_NonSecure else PAS\_Secure;
        when SS\_Realm =>
            r.ipaspace = PAS\_Realm;
        otherwise =>
            // Root security state does not have stage 2 translation
            unreachable;
    end;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBIP_RVA

```
// AArch64_TLBIP_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBIP_RVA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    elsif regime == Regime\_EL2 && ELIsInHost(EL2) then
        regime = Regime\_EL20;
    end;

    var r : TLBIRecord;
    r.op          = TLBIOp\_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt[63:48];
    r.ttl[1:0]     = Xt[38:37];
    r.d64          = r.ttl[1:0] == '00';
    r.d128         = TRUE;

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return; end;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBIP_RVAA

```
// AArch64_TLBIP_RVAA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBIP_RVAA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, level : TLBIlevel, attr_in : TLBIMemAttr,
                        Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    end;

    var r : TLBIRecord;
    r.op          = TLBIOp\_RVAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl[1:0]     = Xt[38:37];
    r.d64          = r.ttl[1:0] == '00';
    r.d128         = TRUE;

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return; end;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```


Library pseudocode for aarch64/functions/tlbi/AArch64_TLBIP_VA

```
// AArch64_TLBIP_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBIP_VA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if regime == Regime\_EL10 && ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    elsif regime == Regime\_EL2 && ELIsInHost(EL2) then
        regime = Regime\_EL20;
    end;

    var r : TLBIRecord;
    r.op      = TLBIOp\_VA;
    r.from_aarch64 = TRUE;
    r.security  = security;
    r.regime    = regime;
    r.vmid      = vmid;
    r.use_vmid  = UseVMID(regime);
    r.level     = level;
    r.attr      = attr;
    r.asid      = Xt[63:48];
    r.ttl       = Xt[47:44];
    r.address   = ZeroExtend{64}(Xt[107:64] :: Zeros{12});
    r.d64       = r.ttl == '00xx';
    r.d128      = TRUE;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBIP_VAA

```
// AArch64_TLBIP_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBIP_VAA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Xt : bits(128))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;
    if ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    end;

    var r : TLBIRecord;
    r.op      = TLBIOp\_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime   = regime;
    r.vmid     = vmid;
    r.use_vmid = UseVMID(regime);
    r.level    = level;
    r.attr     = attr;
    r.ttl      = Xt[47:44];
    r.address   = ZeroExtend{64}(Xt[107:64] :: Zeros{12});
    r.d64      = r.ttl == '00xx';
    r.d128     = TRUE;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_ALL

```
// AArch64_TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch64_TLBI_ALL(security : SecurityState, regime : Regime,
    broadcast_in : Broadcast, attr : TLBIMemAttr, Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op      = TLBIOp\_ALL;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime   = regime;
    r.level    = TLBILevel\_Any;
    r.attr     = attr;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_ASID

```
// AArch64_TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBI_ASID(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
                       broadcast_in : Broadcast, attr_in : TLBMemAttr,
                       Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    end;

    var r : TLBIRecord;
    r.op      = TLBIOp\_ASID;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime   = regime;
    r.vmid     = vmid;
    r.use_vmid = UseVMID(regime);
    r.level    = TLBILevel\_Any;
    r.attr     = attr;
    r.asid     = Xt[63:48];

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_IPAS2

```
// AArch64_TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

func AArch64_TLBI_IPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, level : TLBILevel, attr : TLBIMemAttr,
                        Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op      = TLBIOp\_IPAS2;
    r.from_aarch64 = TRUE;
    r.security  = security;
    r.regime    = regime;
    r.vmid      = vmid;
    r.use_vmid  = TRUE;
    r.level     = level;
    r.attr      = attr;
    r.ttl       = Xt[47:44];
    r.address   = ZeroExtend{64}(Xt[39:0] :: Zeros{12});
    r.d64       = TRUE;
    r.d128      = r.ttl == '00xx';

    case security of
        when SS\_NonSecure =>
            r.ipaspace = PAS\_NonSecure;
        when SS\_Secure =>
            r.ipaspace = if Xt[63] == '1' then PAS\_NonSecure else PAS\_Secure;
        when SS\_Realm =>
            r.ipaspace = PAS\_Realm;
        otherwise =>
            // Root have security state does not have stage 2 translation
            unreachable;
    end;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_PAALL

```
// AArch64_TLBI_PAALL()
// =====
// TLB Invalidate ALL GPT Information.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareability domain.
// The invalidation applies to all TLB entries containing GPT information.

func AArch64_TLBI_PAALL(broadcast : Broadcast)
begin
    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

    var r : TLBIRecord;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op      = TLBIOp\_PAALL;
    r.level   = TLBILevel\_Any;
    r.attr    = TLBI\_AllAttr;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_RIPAS2

```
// AArch64_TLBI_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// broadcast domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.

func AArch64_TLBI_RIPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, level : TLBILevel, attr : TLBIMemAttr,
                        Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op = TLBIOp\_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.use_vmid = TRUE;
    r.level = level;
    r.attr = attr;
    r.ttl[1:0] = Xt[38:37];
    r.d64 = TRUE;
    r.d128 = r.ttl[1:0] == '00';

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return; end;

    case security of
        when SS\_NonSecure =>
            r.ipaspace = PAS\_NonSecure;
        when SS\_Secure =>
            r.ipaspace = if Xt[63] == '1' then PAS\_NonSecure else PAS\_Secure;
        when SS\_Realm =>
            r.ipaspace = PAS\_Realm;
        otherwise =>
            // Root security state does not have stage 2 translation
            unreachable;
    end;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_RPA

```
// AArch64_TLBI_RPA()
// =====
// TLB Range Invalidate GPT Information by PA.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareability domain.
// The invalidation applies to TLB entries containing GPT information relating
// to the indicated physical address range.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries containing GPT information
//                     from all levels of the GPT walk
//     TLBILevel_Last : this applies to TLB entries containing GPT information
//                     from the last level of the GPT walk

func AArch64_TLBI_RPA(level : TLBILevel, Xt : bits(64), broadcast : Broadcast)
begin
    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

    var r : TLBIRecord;
    var range_bits : AddressSize;
    var p : AddressSize;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op = TLBIOp_RPA;
    r.level = level;
    r.attr = TLBI_AllAttr;

    // SIZE field
    case Xt[47:44] of
        when '0000' => range_bits = 12; // 4KB
        when '0001' => range_bits = 14; // 16KB
        when '0010' => range_bits = 16; // 64KB
        when '0011' => range_bits = 21; // 2MB
        when '0100' => range_bits = 25; // 32MB
        when '0101' => range_bits = 29; // 512MB
        when '0110' => range_bits = 30; // 1GB
        when '0111' => range_bits = 34; // 16GB
        when '1000' => range_bits = 36; // 64GB
        when '1001' => range_bits = 39; // 512GB
        otherwise => return; // Reserved encoding, no TLB entries are required to be invalidated
    end;

    // If SIZE selects a range smaller than PGS, then PGS is used instead
    case DecodePGS(GPCCR_EL3().PGS) of
        when PGS_4KB => p = 12;
        when PGS_16KB => p = 14;
        when PGS_64KB => p = 16;
        otherwise => return; // Reserved encoding, no TLB entries are required to be invalidated
    end;

    if range_bits < p then
        range_bits = p;
    end;

    var BaseADDR : bits(NUM_PABITS) = Zeros{};
    case GPCCR_EL3().PGS of
        when '00' => BaseADDR[51:12] = Xt[39:0]; // 4KB
        when '10' => BaseADDR[51:14] = Xt[39:2]; // 16KB
        when '01' => BaseADDR[51:16] = Xt[39:4]; // 64KB
    end;

    if IsFeatureImplemented(FEAT_D128) && AArch64_PAMax() == 56 then
        BaseADDR[55:52] = Xt[43:40]; // Extension to Address
    end;

    // The calculation here automatically aligns BaseADDR to the size of
    // the region specified in SIZE. However, the architecture does not
    // require this alignment and if BaseADDR is not aligned to the region
    // specified by SIZE then no entries are required to be invalidated.
    let range_pbits : integer{} = range_bits;
    let start_addr : bits(NUM_PABITS) = BaseADDR AND NOT ZeroExtend{NUM_PABITS}(Ones{range_pbits});
    let end_addr : bits(NUM_PABITS) = start_addr + ZeroExtend{NUM_PABITS}(Ones{range_pbits});

    // PAspace is not considered in TLBI by PA operations
    r.address = ZeroExtend{64}(start_addr);
    r.end_address = ZeroExtend{64}(end_addr);

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_RVA

```
// AArch64_TLBI_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBI_RVA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if regime == Regime\_EL10 && ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    elsif regime == Regime\_EL2 && ELIsInHost(EL2) then
        regime = Regime\_EL20;
    end;

    var r : TLBIRecord;
    r.op          = TLBIOp\_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level       = level;
    r.attr        = attr;
    r.asid        = Xt[63:48];
    r.ttl[1:0]    = Xt[38:37];
    r.d64         = TRUE;
    r.d128        = r.ttl[1:0] == '00';

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return; end;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_RVAA

```
// AArch64_TLBI_RVAA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBI_RVAA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    end;

    var r : TLBIRecord;
    r.op      = TLBIOp\_RVAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime   = regime;
    r.vmid     = vmid;
    r.use_vmid = UseVMID(regime);
    r.level    = level;
    r.attr     = attr;
    r.ttl[1:0] = Xt[38:37];
    r.d64      = TRUE;
    r.d128     = r.ttl[1:0] == '00';

    var valid : boolean;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return; end;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;
    return;
end;
```


Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_VA

```
// AArch64_TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBI_VA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;
    if regime == Regime\_EL10 && ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    elsif regime == Regime\_EL2 && ELIsInHost(EL2) then
        regime = Regime\_EL20;
    end;

    var r : TLBIRecord;
    r.op      = TLBIOp\_VA;
    r.from_aarch64 = TRUE;
    r.security  = security;
    r.regime    = regime;
    r.vmid      = vmid;
    r.use_vmid  = UseVMID(regime);
    r.level     = level;
    r.attr      = attr;
    r.asid      = Xt[63:48];
    r.ttl       = Xt[47:44];
    r.address    = ZeroExtend{64}(Xt[43:0] :: Zeros{12});
    r.d64       = TRUE;
    r.d128      = r.ttl == '00xx';

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_VAA

```
// AArch64_TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch64_TLBI_VAA(security : SecurityState, regime_in : Regime, vmid_in : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;
    if ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    end;

    var r : TLBIRecord;
    r.op      = TLBIOp\_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime   = regime;
    r.vmid     = vmid;
    r.use_vmid = UseVMID(regime);
    r.level    = level;
    r.attr     = attr;
    r.ttl      = Xt[47:44];
    r.address  = ZeroExtend{64}(Xt[43:0] :: Zeros{12});
    r.d64      = TRUE;
    r.d128     = r.ttl == '00xx';

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_VMALL

```
// AArch64_TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

func AArch64_TLBI_VMALL(security : SecurityState, regime_in : Regime,
                        vmid_in : bits(NUM_VMIDBITS), broadcast_in : Broadcast,
                        attr_in : TLBIMemAttr, Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2, EL1};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);
    var attr      : TLBIMemAttr = attr_in;

    if attr == TLBI\_AllAttr && ExcludeXS() then
        attr = TLBI\_ExcludeXS;
    end;

    var regime : Regime = regime_in;
    var vmid   : bits(NUM_VMIDBITS) = vmid_in;

    if ELIsInHost(EL0) then
        regime = Regime\_EL20;
        vmid   = VMID\_NONE;
    end;

    var r : TLBIRecord;
    r.op      = TLBIOp\_VMALL;
    r.from_aarch64 = TRUE;
    r.security  = security;
    r.regime    = regime;
    r.level     = TLBILevel\_Any;
    r.vmid      = vmid;
    r.use_vmid  = UseVMID(regime);
    r.attr      = attr;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_VMALLS12

```
// AArch64_TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// broadcast domain that match the indicated VMID.

func AArch64_TLBI_VMALLS12(security : SecurityState, regime : Regime,
                           vmid : bits(NUM_VMIDBITS), broadcast_in : Broadcast,
                           attr : TLBIMemAttr, Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    var broadcast : Broadcast = AArch64\_EffectiveBroadcast(broadcast_in);

    var r : TLBIRecord;
    r.op      = TLBIOp\_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security  = security;
    r.regime    = regime;
    r.level     = TLBILevel\_Any;
    r.vmid      = vmid;
    r.use_vmid  = TRUE;
    r.attr      = attr;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/AArch64_TLBI_VMALLWS2

```
// AArch64_TLBI_VMALLWS2()
// =====
// Remove stage 2 dirty state from entries for the indicated translation regime
// with the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID.

func AArch64_TLBI_VMALLWS2(security : SecurityState, regime : Regime, vmid : bits(16),
                           broadcast_in : Broadcast, attr : TLBIMemAttr,
                           Xt : bits(64))
begin
    assert PSTATE.EL IN {EL3, EL2};
    assert regime == Regime EL10;

    if security == SS\_Secure && HaveEL(EL3) && SCR\_EL3().EEL2 == '0' then
        return;
    end;

    var broadcast : Broadcast = broadcast_in;

    var r : TLBIRecord;
    r.op          = TLBIOp\_VMALLWS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel Any;
    r.vmid        = vmid;
    r.use_vmid     = TRUE;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r); end;

    return;
end;
```

Library pseudocode for aarch64/functions/tlbi/DecodeTLBITG

```
// DecodeTLBITG()
// =====
// Decode translation granule size in TLBI range instructions

func DecodeTLBITG(tg : bits(2)) => TGx
begin
    case tg of
        when '01' => return TGx\_4KB;
        when '10' => return TGx\_16KB;
        when '11' => return TGx\_64KB;
    end;
end;
```

Library pseudocode for aarch64/functions/tlbi/GPTTLBIMatch

```
// GPTTLBIMatch()
// =====
// Determine whether the GPT TLB entry lies within the scope of invalidation

func GPTTLBIMatch(tlbi : TLBIRecord, gpt_entry : GPTEntry) => boolean
begin
    assert tlbi.op IN {TLBIOp\_RPA, TLBIOp\_PAALL};

    var match : boolean;
    let entry_size_mask : bits(64) = ZeroExtend{1}(Ones{gpt_entry.size});
    let entry_end_address : bits(64) = (ZeroExtend{64}(gpt_entry.pa[NUM\_PABITS-1:0]) OR
                                         entry_size_mask[NUM\_PABITS-1:0]);
    let entry_start_address : bits(64) = (ZeroExtend{64}(gpt_entry.pa[NUM\_PABITS-1:0]) AND NOT
                                         entry_size_mask[NUM\_PABITS-1:0]);

    case tlbi.op of
        when TLBIOp\_RPA =>
            match = ((UInt(tlbi.address[NUM\_PABITS-1:0]) <=
                      UInt(entry_end_address[NUM\_PABITS-1:0])) &&
                     (UInt(tlbi.end_address[NUM\_PABITS-1:0]) >
                      UInt(entry_start_address[NUM\_PABITS-1:0])) &&
                     (tlbi.level == TLBILevel Any || !gpt_entry.istable));
        when TLBIOp\_PAALL =>
            match = TRUE;
    end;

    return match;
end;
```

Library pseudocode for aarch64/functions/tlbi/HasLargeAddress

```
// HasLargeAddress()
// =====
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

func HasLargeAddress(regime : Regime) => boolean
begin
    if !IsFeatureImplemented(FEAT_LPA2) then
        return FALSE;
    end;
    case regime of
        when Regime\_EL3 =>
            return TCR_EL3().DS == '1';
        when Regime\_EL2 =>
            return TCR_EL2().DS == '1';
        when Regime\_EL20 =>
            return TCR_EL2().DS == '1';
        when Regime\_EL10 =>
            return TCR_EL1().DS == '1';
        otherwise =>
            unreachable;
    end;
end;
```

Library pseudocode for aarch64/functions/tlbi/ResTLBIRTTL

```
// ResTLBIRTTL()
// =====
// Determine whether the TTL field in TLBI instructions that do apply
// to a range of addresses contains a reserved value

func ResTLBIRTTL(tg : bits(2), ttl : bits(2)) => boolean
begin
    case ttl of
        when '00' => return TRUE;
        when '01' => return DecodeTLBITG(tg) == TGx\_16KB && !IsFeatureImplemented(FEAT_LPA2);
        otherwise => return FALSE;
    end;
end;
```

Library pseudocode for aarch64/functions/tlbi/ResTLBITTL

```
// ResTLBITTL()
// =====
// Determine whether the TTL field in TLBI instructions that do not apply
// to a range of addresses contains a reserved value

func ResTLBITTL(ttl : bits(4)) => boolean
begin
    case ttl of
        when '00xx' => return TRUE;
        when '0100' => return !IsFeatureImplemented(FEAT_LPA2);
        when '1000' => return TRUE;
        when '1001' => return !IsFeatureImplemented(FEAT_LPA2);
        when '1100' => return TRUE;
        otherwise => return FALSE;
    end;
end;
```

Library pseudocode for aarch64/functions/tlbi/TGBits

```
// TGBits()
// =====
// Return the number of least-significant address bits within a single Translation Granule.

func TGBits(tg : bits(2)) => AddressSize
begin
    case tg of
        when '01' => return 12; // 4KB
        when '10' => return 14; // 16KB
        when '11' => return 16; // 64KB
        otherwise =>
            unreachable;
    end;
end;
```



```

// TLBIMatch()
// =====
// Determine whether the TLB entry lies within the scope of invalidation

func TLBIMatch(tlbi : TLBIRecord, tlb_entry : TLBRecord) => boolean
begin
    var match : boolean;
    let entry_block_mask : bits(64) = ZeroExtend{}(Ones{tlb_entry.blocksize});
    var entry_end_address : bits(64) = tlb_entry.context.ia OR entry_block_mask;
    var entry_start_address : bits(64) = tlb_entry.context.ia AND NOT entry_block_mask;
    case tlbi.op of
        when TLBIOp\_DALL, TLBIOp\_IALL =>
            match = (tlbi.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime &&
                    tlb_entry.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid));
        when TLBIOp\_DASID, TLBIOp\_IASID =>
            match = (tlb_entry.context.includes_s1 &&
                    tlb_entry.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime &&
                    tlb_entry.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid) &&
                    (UseASID(tlb_entry.context) && tlb_entry.context.nG == '1' &&
                     tlb_entry.asid == tlb_entry.context.asid));
        when TLBIOp\_DVA, TLBIOp\_IVA =>
            var regime_match : boolean;
            var context_match : boolean;
            var address_match : boolean;
            var level_match : boolean;
            regime_match = (tlb_entry.context.includes_s1 &&
                           tlb_entry.security == tlb_entry.context.ss &&
                           tlb_entry.regime == tlb_entry.context.regime);
            context_match = (tlb_entry.use_vmid == tlb_entry.context.use_vmid &&
                            (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid) &&
                            (!UseASID(tlb_entry.context) || tlb_entry.asid == tlb_entry.context.asid ||
                             tlb_entry.context.nG == '0'));
            let addr_lsb : integer{} = tlb_entry.blocksize;
            address_match = tlb_entry.address[55:addr_lsb] == tlb_entry.context.ia[55:addr_lsb];
            level_match = (tlbi.level == TLBIlevel\_Any || !tlb_entry.walkstate.istable);
            match = regime_match && context_match && address_match && level_match;
        when TLBIOp\_ALL =>
            let relax_regime : boolean = (tlbi.from_aarch64 &&
                                           tlb_entry.regime IN {Regime\_EL20, Regime\_EL2} &&
                                           tlb_entry.context.regime IN {Regime\_EL20, Regime\_EL2});
            match = (tlbi.security == tlb_entry.context.ss &&
                    (tlbi.regime == tlb_entry.context.regime || relax_regime));
        when TLBIOp\_ASID =>
            match = (tlb_entry.context.includes_s1 &&
                    tlb_entry.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime &&
                    tlb_entry.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid) &&
                    (UseASID(tlb_entry.context) && tlb_entry.context.nG == '1' &&
                     tlb_entry.asid == tlb_entry.context.asid));
        when TLBIOp\_IPAS2, TLBIOp\_IPAS2 =>
            let addr_lsb : integer{} = tlb_entry.blocksize;
            match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
                    tlb_entry.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime &&
                    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid) &&
                    tlb_entry.ipaspace == tlb_entry.context.ipaspace &&
                    (tlbi.address[NUM\_PABITS-1:addr_lsb] ==
                     tlb_entry.context.ia[NUM\_PABITS-1:addr_lsb]) &&
                    (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                     DecodeTLBITG(tlbi.ttl[3:2]) == tlb_entry.context.tg &&
                     UInt(tlbi.ttl[1:0]) == tlb_entry.walkstate.level)
                    ) &&
                    ((tlbi.d128 && tlb_entry.context.isd128) ||
                     (tlbi.d64 && !tlb_entry.context.isd128) ||
                     (tlbi.d64 && tlb_entry.d128)) &&
                    (tlbi.level == TLBIlevel\_Any || !tlb_entry.walkstate.istable));
        when TLBIOp\_VAA, TLBIOp\_VAA =>
            let addr_lsb : integer{} = tlb_entry.blocksize;
            match = (tlb_entry.context.includes_s1 &&
                    tlb_entry.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime &&
                    tlb_entry.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid) &&
                    tlb_entry.address[55:addr_lsb] == tlb_entry.context.ia[55:addr_lsb] &&
                    (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                     DecodeTLBITG(tlbi.ttl[3:2]) == tlb_entry.context.tg &&
                     UInt(tlbi.ttl[1:0]) == tlb_entry.walkstate.level)
                    ) &&
                    ((tlbi.d128 && tlb_entry.context.isd128) ||
                     (tlbi.d64 && !tlb_entry.context.isd128) ||
                     (tlbi.d64 && tlb_entry.d128)) &&
                    (tlbi.level == TLBIlevel\_Any || !tlb_entry.walkstate.istable));
        when TLBIOp\_VA, TLBIOp\_VA =>
            let addr_lsb : integer{} = tlb_entry.blocksize;

```

```

match = (tlb_entry.context.includes_s1 &&
  tlb_i.security == tlb_entry.context.ss &&
  tlb_i.regime == tlb_entry.context.regime &&
  tlb_i.use_vmid == tlb_entry.context.use_vmid &&
  (!tlb_entry.context.use_vmid || tlb_i.vmid == tlb_entry.context.vmid) &&
  (!UseASID(tlb_entry.context) || tlb_i.asid == tlb_entry.context.asid ||
    tlb_entry.context.nG == '0') &&
  tlb_i.address[55:addr_lsb] == tlb_entry.context.ia[55:addr_lsb] &&
  (!tlb_i.from_aarch64 || ResTLBITTL(tlb_i.ttl) || (
    DecodeTLBITG(tlb_i.ttl[3:2]) == tlb_entry.context.tg &&
    UInt(tlb_i.ttl[1:0]) == tlb_entry.walkstate.level)
  ) &&
  ((tlb_i.d128 && tlb_entry.context.isd128) ||
  (tlb_i.d64 && !tlb_entry.context.isd128) ||
  (tlb_i.d64 && tlb_i.d128)) &&
  (tlb_i.level == TLBIlevel Any || !tlb_entry.walkstate.istable));
when TLBIop VMALL =>
  match = (tlb_entry.context.includes_s1 &&
    tlb_i.security == tlb_entry.context.ss &&
    tlb_i.regime == tlb_entry.context.regime &&
    tlb_i.use_vmid == tlb_entry.context.use_vmid &&
    (!tlb_entry.context.use_vmid || tlb_i.vmid == tlb_entry.context.vmid));
when TLBIop VMALLS12 =>
  match = (tlb_i.security == tlb_entry.context.ss &&
    tlb_i.regime == tlb_entry.context.regime &&
    (!tlb_entry.context.use_vmid || tlb_i.vmid == tlb_entry.context.vmid));
when TLBIop RIPAS2, TLBIop RIPAS2 =>
  match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
    tlb_i.security == tlb_entry.context.ss &&
    tlb_i.regime == tlb_entry.context.regime &&
    (!tlb_entry.context.use_vmid || tlb_i.vmid == tlb_entry.context.vmid) &&
    tlb_i.ipaspace == tlb_entry.context.ipaspace &&
    (tlb_i.tg != '00' && DecodeTLBITG(tlb_i.tg) == tlb_entry.context.tg) &&
    (!tlb_i.from_aarch64 || ResTLBITTL(tlb_i.tg, tlb_i.ttl[1:0]) ||
      UInt(tlb_i.ttl[1:0]) == tlb_entry.walkstate.level) &&
    ((tlb_i.d128 && tlb_entry.context.isd128) ||
    (tlb_i.d64 && !tlb_entry.context.isd128) ||
    (tlb_i.d64 && tlb_i.d128)) &&
    (UInt(tlb_i.address[NUM_PABITS-1:0]) <=
      UInt(entry_end_address[NUM_PABITS-1:0])) &&
    (UInt(tlb_i.end_address[NUM_PABITS-1:0]) >
      UInt(entry_start_address[NUM_PABITS-1:0])));
when TLBIop RVAA, TLBIop RVAA =>
  match = (tlb_entry.context.includes_s1 &&
    tlb_i.security == tlb_entry.context.ss &&
    tlb_i.regime == tlb_entry.context.regime &&
    tlb_i.use_vmid == tlb_entry.context.use_vmid &&
    (!tlb_entry.context.use_vmid || tlb_i.vmid == tlb_entry.context.vmid) &&
    (tlb_i.tg != '00' && DecodeTLBITG(tlb_i.tg) == tlb_entry.context.tg) &&
    (!tlb_i.from_aarch64 || ResTLBITTL(tlb_i.tg, tlb_i.ttl[1:0]) ||
      UInt(tlb_i.ttl[1:0]) == tlb_entry.walkstate.level) &&
    ((tlb_i.d128 && tlb_entry.context.isd128) ||
    (tlb_i.d64 && !tlb_entry.context.isd128) ||
    (tlb_i.d64 && tlb_i.d128)) &&
    UInt(tlb_i.address[55:0]) <= UInt(entry_end_address[55:0]) &&
    UInt(tlb_i.end_address[55:0]) > UInt(entry_start_address[55:0]));
when TLBIop RVA, TLBIop RVA =>
  match = (tlb_entry.context.includes_s1 &&
    tlb_i.security == tlb_entry.context.ss &&
    tlb_i.regime == tlb_entry.context.regime &&
    tlb_i.use_vmid == tlb_entry.context.use_vmid &&
    (!tlb_entry.context.use_vmid || tlb_i.vmid == tlb_entry.context.vmid) &&
    (!UseASID(tlb_entry.context) || tlb_i.asid == tlb_entry.context.asid ||
      tlb_entry.context.nG == '0') &&
    (tlb_i.tg != '00' && DecodeTLBITG(tlb_i.tg) == tlb_entry.context.tg) &&
    (!tlb_i.from_aarch64 || ResTLBITTL(tlb_i.tg, tlb_i.ttl[1:0]) ||
      UInt(tlb_i.ttl[1:0]) == tlb_entry.walkstate.level) &&
    ((tlb_i.d128 && tlb_entry.context.isd128) ||
    (tlb_i.d64 && !tlb_entry.context.isd128) ||
    (tlb_i.d64 && tlb_i.d128)) &&
    UInt(tlb_i.address[55:0]) <= UInt(entry_end_address[55:0]) &&
    UInt(tlb_i.end_address[55:0]) > UInt(entry_start_address[55:0]));
when TLBIop RPA =>
  let addr = tlb_entry.walkstate.baseaddress.address;
  entry_end_address[NUM_PABITS-1:0] = (addr[NUM_PABITS-1:0] OR
    entry_block_mask[NUM_PABITS-1:0]);
  entry_start_address[NUM_PABITS-1:0] = (addr[NUM_PABITS-1:0] AND
    NOT entry_block_mask[NUM_PABITS-1:0]);
  match = (tlb_entry.context.includes_gpt &&
    (UInt(tlb_i.address[NUM_PABITS-1:0]) <=
      UInt(entry_end_address[NUM_PABITS-1:0])) &&
    (UInt(tlb_i.end_address[NUM_PABITS-1:0]) >
      UInt(entry_start_address[NUM_PABITS-1:0])));
when TLBIop PAALL =>
  match = tlb_entry.context.includes_gpt;
end;

if tlb_i.attr == TLBI ExcludeXS && tlb_entry.context.xs == '1' then
  match = FALSE;

```



```

end;

return match;

end;

```

Library pseudocode for aarch64/functions/tlbi/TLBIPRange

```

// TLBIPRange()
// =====
// Extract the input address range information from encoded Xt.

func TLBIPRange(regime : Regime, Xt : bits(128)) => (boolean, bits(2), bits(64), bits(64))
begin
    let valid : boolean = TRUE;
    var start_address : bits(64) = Zeros{};
    var end_address : bits(64) = Zeros{};

    let tg : bits(2) = Xt[47:46];
    let scale : integer = UInt(Xt[45:44]);
    let num : integer = UInt(Xt[43:39]);

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);
    end;

    let tg_bits : AddressSize = TGBits(tg);
    // The more-significant bits of the start_address is not updated,
    // as they are not used when performing address matching in TLB
    start_address[55:tg_bits] = Xt[107:64+(tg_bits-12)];

    let range : integer = (num+1) << (5*scale + 1 + tg_bits);
    end_address = start_address + range[63:0];

    if end_address[55] != start_address[55] then
        // overflow, saturate it
        end_address = Replicate{9}(start_address[55]) :: Ones{55};
    end;

    return (valid, tg, start_address, end_address);
end;

```

Library pseudocode for aarch64/functions/tlbi/TLBIRange

```
// TLBIRange()
// =====
// Extract the input address range information from encoded Xt.

func TLBIRange(regime : Regime, Xt : bits(64)) => (boolean, bits(2), bits(64), bits(64))
begin
    let valid : boolean = TRUE;
    var start_address : bits(64) = Zeros{};
    var end_address : bits(64) = Zeros{};

    let tg : bits(2) = Xt[47:46];
    let scale : integer = UInt(Xt[45:44]);
    let num : integer = UInt(Xt[43:39]);
    var tg_bits : integer;

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);
    end;

    case tg of
        when '01' => // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start_address[52:16] = Xt[36:0];
                start_address[63:53] = Replicate{11}(Xt[36]);
            else
                start_address[48:12] = Xt[36:0];
                start_address[63:49] = Replicate{15}(Xt[36]);
            end;
        when '10' => // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start_address[52:16] = Xt[36:0];
                start_address[63:53] = Replicate{11}(Xt[36]);
            else
                start_address[50:14] = Xt[36:0];
                start_address[63:51] = Replicate{13}(Xt[36]);
            end;
        when '11' => // 64KB
            tg_bits = 16;
            start_address[52:16] = Xt[36:0];
            start_address[63:53] = Replicate{11}(Xt[36]);
        otherwise =>
            unreachable;
    end;

    let range : integer = (num+1) << (5*scale + 1 + tg_bits);
    end_address = start_address + range[63:0];

    if IsFeatureImplemented(FEAT_LVA3) && end_address[56] != start_address[56] then
        // overflow, saturate it
        end_address = Replicate{8}(start_address[56]) :: Ones{56};
    elsif end_address[52] != start_address[52] then
        // overflow, saturate it
        end_address = Replicate{12}(start_address[52]) :: Ones{52};
    end;

    return (valid, tg, start_address, end_address);
end;
```

Library pseudocode for aarch64/functions/vbitop/VBitOp

```
// VBitOp
// =====
// Vector bit select instruction types.

type VBitOp of enumeration {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

Library pseudocode for aarch64/translation/attrs/AArch64_MAIRAttr

```
// AArch64_MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR

func AArch64_MAIRAttr(index : integer, mair2 : MAIRType, mair : MAIRType) => bits(8)
begin
    assert (index < 8 || (IsFeatureImplemented(FEAT_AIE) && (index < 16)));
    if (index > 7) then
        return mair2[(index-8)*:8]; // Read from LSB at MAIR2
    else
        return mair[index*:8];
    end;
end;
```

Library pseudocode for aarch64/translation/debug/AArch64_CheckBreakpoint

```
// AArch64_CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

func AArch64_CheckBreakpoint(fault_in : FaultRecord, vaddress : bits(64),
                             accdesc : AccessDescriptor, size : integer) => FaultRecord
begin
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    var fault : FaultRecord = fault_in;
    var match : boolean = FALSE;
    var addr_match_bp : boolean = FALSE; // Default assumption that all address match
                                         // breakpoints are inactive or disabled.
    var addr_mismatch_bp : boolean = FALSE; // Default assumption that all address mismatch
                                         // breakpoints are inactive or disabled.
    var addr_match : boolean = FALSE;
    var addr_mismatch : boolean = TRUE; // Default assumption that the given virtual address
                                         // is outside the range of all address mismatch
                                         // breakpoints.
    var ctxt_match : boolean = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1 do
        let brkptinfo : BreakpointInfo = AArch64\_BreakpointMatch(i, vaddress, accdesc, size);
        if brkptinfo.bptype == BreakpointType\_AddrMatch then
            addr_match_bp = TRUE;
            addr_match = addr_match || brkptinfo.match;
        elseif brkptinfo.bptype == BreakpointType\_AddrMismatch then
            addr_mismatch_bp = TRUE;
            addr_mismatch = addr_mismatch && !brkptinfo.match;
        elseif brkptinfo.bptype == BreakpointType\_CtxtMatch then
            ctxt_match = ctxt_match || brkptinfo.match;
        end;
    end;
    if addr_match_bp && addr_mismatch_bp then
        match = addr_match && addr_mismatch;
    else
        match = (addr_match_bp && addr_match) || (addr_mismatch_bp && addr_mismatch);
    end;

    match = match || ctxt_match;

    if match then
        fault.statuscode = Fault\_Debug;
        fault.vaddress = vaddress;
        if HaltOnBreakpointOrWatchpoint() then
            let reason : bits(6) = DebugHalt\_Breakpoint;
            Halt(reason);
        end;
    end;

    return fault;
end;
```

Library pseudocode for aarch64/translation/debug/AArch64_CheckDebug

```
// AArch64_CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

func AArch64_CheckDebug(vaddress : bits(64), accdesc : AccessDescriptor,
                        size : integer) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, vaddress);
    var generate_exception : boolean;

    let d_side : boolean = IsWatchpointableAccess(accdesc);
    let i_side : boolean = (accdesc.acctype == AccessType\_IFETCH);
    if accdesc.acctype == AccessType\_NV2 then
        let mask : bit = '0';
        let ss : SecurityState = CurrentSecurityState();
        generate_exception = (AArch64\_GenerateDebugExceptionsFrom(EL2, ss, mask) &&
                             MDSCR\_EL1().MDE == '1');
    else
        generate_exception = AArch64\_GenerateDebugExceptions() && MDSCR\_EL1().MDE == '1';
    end;
    let halt : boolean = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64\_CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch64\_CheckBreakpoint(fault, vaddress, accdesc, size);
        end;
    end;

    return fault;
end;
```



```

// AArch64_CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

func AArch64_CheckWatchpoint(fault_in : FaultRecord, vaddress_in : bits(64),
                             accdesc : AccessDescriptor, size_in : integer) => FaultRecord
begin
    assert !ELUsingAArch32\(S1TranslationRegime\(\)\);
    var fault : FaultRecord = fault_in;
    var fault_match : FaultRecord = fault_in;
    var fault_mismatch : FaultRecord = fault_in;
    var vaddress : bits(64) = vaddress_in;
    var size : integer = size_in;
    var rounded_match : boolean = FALSE;
    let original_vaddress : bits(64) = vaddress;
    let original_size : integer = size;
    var addr_match_wp : boolean = FALSE; // Default assumption that all address
                                         // match watchpoints are inactive or disabled.
    var addr_mismatch_wp : boolean = FALSE; // Default assumption that all address mismatch
                                         // watchpoints are inactive or disabled.
    var addr_match : boolean = FALSE;
    var addr_mismatch : boolean = TRUE; // Default assumption that the given virtual address is
                                         // outside the range of all address mismatch watchpoints

    // For memory accesses of below type
    // - Contiguous SVE access
    // - SME access
    // - SIMD&FP access when the PE is in Streaming SVE mode
    // each call to this function is such that:
    // - the lowest accessed address is rounded down to the nearest multiple of 16 bytes
    // - the highest accessed address is rounded up to the nearest multiple of 16 bytes
    // Since the WPF field is set if the implementation does rounding, regardless of true or
    // false match, it would be acceptable to return TRUE for either/both of the first and last
    // access.
    if IsRelaxedWatchpointAccess(accdesc) then
        var upper_vaddress : integer = UInt(original_vaddress) + original_size;
        if ConstrainUnpredictableBool\(Unpredictable\_16BYTEROUNDEDDOWNACCESS\) then
            vaddress = AlignDownSize{64}(vaddress, 16);
            rounded_match = TRUE;
        end;
        if ConstrainUnpredictableBool\(Unpredictable\_16BYTEROUNDEDUPACCESS\) then
            upper_vaddress = AlignUpSize(upper_vaddress, 16);
            rounded_match = TRUE;
        end;
        size = upper_vaddress - UInt(vaddress);
    end;

    for i = 0 to NumWatchpointsImplemented\(\) - 1 do
        let watchptinfo : WatchpointInfo = AArch64\_WatchpointMatch(i, vaddress, size, accdesc);
        if watchptinfo.wptype == WatchpointType AddrMatch then
            addr_match_wp = TRUE;
            addr_match = addr_match || watchptinfo.value_match;
            if watchptinfo.value_match then
                fault_match.statuscode = Fault\_Debug;
                if DBGWCR_EL1(i).LSC[0] == '1' && accdesc.read then
                    fault_match.write = FALSE;
                elsif DBGWCR_EL1(i).LSC[1] == '1' && accdesc.write then
                    fault_match.write = TRUE;
                end;
                fault_match.watchptinfo = watchptinfo;
            end;
        elsif watchptinfo.wptype == WatchpointType AddrMismatch then
            addr_mismatch_wp = TRUE;
            addr_mismatch = addr_mismatch && !watchptinfo.value_match;
            if !watchptinfo.value_match then
                fault_mismatch.statuscode = Fault\_Debug;
                if DBGWCR_EL1(i).LSC[0] == '1' && accdesc.read then
                    fault_mismatch.write = FALSE;
                elsif DBGWCR_EL1(i).LSC[1] == '1' && accdesc.write then
                    fault_mismatch.write = TRUE;
                end;
                fault_mismatch.watchptinfo = watchptinfo;
            end;
        end;
    end;
    if ((addr_match_wp && addr_mismatch_wp && addr_match && addr_mismatch) ||
        (addr_match_wp && !addr_mismatch_wp && addr_match)) then
        fault = fault_match;
    elsif !addr_match_wp && addr_mismatch_wp && addr_mismatch then
        fault = fault_mismatch;
    end;
    fault.vaddress = vaddress;
    fault.watchptinfo.maybe_false_match = rounded_match;
    if (fault.statuscode == Fault\_Debug && HaltOnBreakpointOrWatchpoint\(\) &&
        !accdesc.nonfault && !(accdesc.firstfault && !accdesc.first)) then
        let reason : bits(6) = DebugHalt\_Watchpoint;
        EDWAR() = fault.vaddress;
    end;
end;

```

```

    let is_async : boolean = FALSE;
    Halt(reason, is_async, fault);
end;
return fault;
end;

```

Library pseudocode for aarch64/translation/hdbss/AppendToHDBSS

```

// AppendToHDBSS ()
// =====
// Appends an entry to the HDBSS when the dirty state of a stage 2 descriptor is updated
// from writable-clean to writable-dirty by hardware.

func AppendToHDBSS(fault_in : FaultRecord, ipa_in : FullAddress, accdesc : AccessDescriptor,
    walkparams : S2TTWParams, level : integer) => FaultRecord
begin
    assert CanAppendToHDBSS();

    var fault : FaultRecord = fault_in;
    var ipa : FullAddress = ipa_in;
    let hdbss_size : integer{} = UInt(HDBSSBR_EL2().SZ) as integer{0..9};

    var hdbss_addrdesc : AddressDescriptor;

    var baddr : bits(NUM_PABITS) = HDBSSBR_EL2().BADDR[NUM_PABITS-13 : 0] :: Zeros{12};
    baddr[11 + hdbss_size : 12] = Zeros{hdbss_size};

    hdbss_addrdesc.paddress.address = baddr + (8 * UInt(HDBSSPROD_EL2().INDEX));
    let nse2 : bit = '0'; // NSE2 has the Effective value of 0 within a PE.
    hdbss_addrdesc.paddress.paspace = DecodePASpace(nse2, EffectiveSCR_EL3_NSE(),
        EffectiveSCR_EL3_NS());

    // Accesses to the HDBSS use the same memory attributes as used for stage 2 translation walks.
    hdbss_addrdesc.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    let hdbss_access : AccessDescriptor = CreateAccDescHDBSS(accdesc);
    hdbss_addrdesc.mecid = AArch64_S2TTWalkMECID(walkparams.emec, accdesc.ss);

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(hdbss_addrdesc, hdbss_access);

        if fault.gpcf.gpf != GPCF_None then
            if (ImpDefBool(
                "GPC fault on HDBSS write reported in HDBSSPROD_EL2")) then
                HDBSSPROD_EL2().FSC = '101000';
            else
                fault.statuscode = Fault_GPCFOnWalk;
                fault.paddress = hdbss_addrdesc.paddress;
                fault.level = level;
                fault.gpcfs2walk = TRUE;
                fault.hdbssf = TRUE;
            end;

            return fault;
        end;
    end;

    // The reported IPA must be aligned to the size of the translation.
    let lsb : AddressSize = TranslationSize(walkparams.dl28, walkparams.tgx, level);
    ipa.address = ipa.address[NUM_PABITS-1:lsb] :: Zeros{lsb};
    var hdbss_entry : bits(64) = CreateHDBSSEntry(ipa, hdbss_access.ss, level);

    if walkparams.ee == '1' then
        hdbss_entry = BigEndianReverse{64}(hdbss_entry);
    end;

    let memstatus : PhysMemRetStatus = PhysMemWrite{64}(hdbss_addrdesc, hdbss_access,
        hdbss_entry);

    if IsFault(memstatus) then
        if (ImpDefBool(
            "External Abort on HDBSS write reported in HDBSSPROD_EL2")) then
            HDBSSPROD_EL2().FSC = '010000';
        else
            let iswrite : boolean = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, hdbss_addrdesc,
                hdbss_access, 8, fault);

            fault.level = level;
            fault.hdbssf = TRUE;
        end;
    else
        HDBSSPROD_EL2().INDEX = HDBSSPROD_EL2().INDEX + 1;
    end;

    return fault;
end;

```

Library pseudocode for aarch64/translation/hdbss/CanAppendToHDBSS

```
// CanAppendToHDBSS()
// =====
// Return TRUE if HDBSS can be appended.

readonly func CanAppendToHDBSS() => boolean
begin
    if !IsFeatureImplemented(FEAT_HDBSS) then
        return FALSE;
    end;
    assert EL2Enabled\(\);
    // The PE cannot append entries to the HDBSS if HDBSSPROD_EL2.FSC is
    // any other value than 0b000000, or HDBSS buffer is full.

    if ((UInt(HDBSSPROD_EL2().INDEX) >= ((2 ^ (UInt(HDBSSBR_EL2().SZ) + 12)) DIV 8)) ||
        (HDBSSPROD_EL2().FSC != '000000')) then
        return FALSE;
    else
        return TRUE;
    end;
end;
```

Library pseudocode for aarch64/translation/hdbss/CreateHDBSSEntry

```
// CreateHDBSSEntry()
// =====
// Returns a HDBSS entry.

func CreateHDBSSEntry(ipa : FullAddress, ss : SecurityState, level : integer) => bits(64)
begin
    let ns_ipa : bit = if ss == SS Secure && ipa.paspace == PAS NonSecure then '1' else '0';
    return ZeroExtend{64}(ipa.address[NUM\_PABITS-1:12] ::
        ns_ipa :: Zeros{7} :: level[2:0] :: '1');
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_IASize

```
// AArch64_IASize()
// =====
// Retrieve the number of bits containing the input address

func AArch64_IASize(txsz : bits(6)) => AddressSize
begin
    return (64 - UInt(txsz)) as AddressSize;
end;
```


Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_NextTableBase

```
// AArch64_NextTableBase()
// =====
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

func AArch64_NextTableBase(N)(descriptor : bits(N), dl28 : bit, skl : bits(2),
    ds : bit, tgx : TGx) => bits(NUM\_PABITS)

begin
    var tablebase : bits(NUM\_PABITS) = Zeros{};
    let granulebits : AddressSize = TGxGranuleBits(tgx);
    var tablesize : integer;

    if dl28 == '1' then
        let descsize2 : integer = 4;
        let stride : integer = granulebits - descsize2;
        tablesize = stride*(1 + UInt(skl)) + descsize2;
    else
        tablesize = granulebits;
    end;

    case tgx of
        when TGx\_4KB => tablebase[47:12] = descriptor[47:12];
        when TGx\_16KB => tablebase[47:14] = descriptor[47:14];
        when TGx\_64KB => tablebase[47:16] = descriptor[47:16];
    end;

    tablebase = AlignDownP2(NUM\_PABITS)(tablebase, tablesize as integer{0..NUM\_PABITS});

    if dl28 == '1' then
        tablebase[55:48] = descriptor[55:48];
    elsif tgx == TGx\_64KB && (AArch64\_PAMax() >= 52 ||
        ImpDefBool("descriptor[15:12] for 64KB granule are OA[51:48]")) then
        tablebase[51:48] = descriptor[15:12];
    elsif ds == '1' then
        tablebase[51:48] = descriptor[9:8]::descriptor[49:48];
    end;

    return tablebase;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_PhysicalAddressSize

```
// AArch64_PhysicalAddressSize()
// =====
// Retrieve the number of bits bounding the physical address

func AArch64_PhysicalAddressSize(dl28 : bit, ds : bit, encoded_ps : bits(3),
    tgx : TGx) => AddressSize

begin
    var ps : integer;
    var max_ps : integer;

    case encoded_ps of
        when '000' => ps = 32;
        when '001' => ps = 36;
        when '010' => ps = 40;
        when '011' => ps = 42;
        when '100' => ps = 44;
        when '101' => ps = 48;
        when '110' => ps = 52;
        when '111' => ps = 56;
    end;

    if dl28 == '1' then
        max_ps = AArch64\_PAMax();
    elsif IsFeatureImplemented(FEAT_LPA) && (tgx == TGx\_64KB || ds == '1') then
        max_ps = Min(52, AArch64\_PAMax());
    else
        max_ps = Min(48, AArch64\_PAMax());
    end;

    return Min(ps, max_ps) as AddressSize;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S1LeafBase

```
// AArch64_S1LeafBase()
// =====
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

func AArch64_S1LeafBase(N)(descriptor : bits(N), walkparams : S1TTWParams,
    level : integer) => bits(NUM\_PABITS)

begin
    var leafbase : bits(NUM\_PABITS) = Zeros{};

    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer = granulebits - descsize2;
    let leafsize : integer = granulebits + stride * (FINAL\_LEVEL - level);

    leafbase[47:0] = AlignDownP2{48}(descriptor[47:0], leafsize as integer{0..48});

    if walkparams.d128 == '1' then
        leafbase[55:48] = descriptor[55:48];
    elseif walkparams.tgx == TGx\_64KB && (AArch64\_PAMax() >= 52 ||
        ImpDefBool("descriptor[15:12] for 64KB granule are OA[51:48]")) then
        leafbase[51:48] = descriptor[15:12];
    elseif walkparams.ds == '1' then
        leafbase[51:48] = descriptor[9:8,49:48];
    end;

    return leafbase;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S1SLTTEEntryAddress

```
// AArch64_S1SLTTEEntryAddress()
// =====
// Compute the first stage 1 translation table descriptor address within the
// table pointed to by the base at the start level

func AArch64_S1SLTTEEntryAddress(level : integer, walkparams : S1TTWParams,
    ia : bits(64), tablebase : FullAddress) => FullAddress

begin
    // Input Address size
    let iasize : AddressSize = AArch64\_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{} = granulebits - descsize2;
    let levels : integer = FINAL\_LEVEL - level;

    var index : bits(NUM\_PABITS);
    let lsb : AddressSize = (levels*stride + granulebits) as AddressSize;
    let msb : AddressSize = (iasize - 1) as AddressSize;
    index = ZeroExtend(NUM\_PABITS)(ia[msb:lsb]::Zeros{descsize2});

    var descaddress : FullAddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S1StartLevel

```
// AArch64_S1StartLevel()
// =====
// Compute the initial lookup level when performing a stage 1 translation
// table walk

func AArch64_S1StartLevel(walkparams : S1TTWParams) => integer

begin
    // Input Address size
    let iasize : AddressSize = AArch64\_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer = granulebits - descsize2;
    var slstartlevel : integer = FINAL\_LEVEL - ((iasize-1) - granulebits) DIVRM stride);
    if walkparams.d128 == '1' then
        slstartlevel = slstartlevel + UInt(walkparams.sk1);
    end;
    return slstartlevel;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S1TTBaseAddress

```
// AArch64_S1TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 1

func AArch64_S1TTBaseAddress{N}(walkparams : S1TTWParams,
                                regime : Regime, ttbr : bits(N)) => bits(NUM_PABITS)
begin
    var tablebase : bits(NUM_PABITS) = Zeros{};

    // Input Address size
    let iasize : AddressSize = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{} = granulebits - descsize2;
    let startlevel : integer = AArch64_S1StartLevel(walkparams);
    let levels : integer = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    var tsize : integer = (iasize - (levels*stride + granulebits)) + descsize2;

    if walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if regime == Regime_EL3 then
            tablebase[55:5] = ttbr[55:5];
        else
            tablebase[55:5] = ttbr[87:80]::ttbr[47:5];
        end;
    elseif walkparams.ds == '1' || (walkparams.tgx == TGx_64KB && walkparams.ps == '110' &&
        (IsFeatureImplemented(FEAT_LPA) ||
        ImpDefBool("BADDR expresses 52 bits for 64KB granule"))) then
        tsize = Max(tsize, 6);
        tablebase[51:6] = ttbr[5:2]::ttbr[47:6];
    else
        tablebase[47:1] = ttbr[47:1];
    end;
    tablebase = AlignDownP2{NUM_PABITS}(tablebase, tsize as integer{0..NUM_PABITS});
    return tablebase;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S1TTEntryAddress

```
// AArch64_S1TTEntryAddress()
// =====
// Compute translation table descriptor address within the table pointed to by
// the table base

func AArch64_S1TTEntryAddress{N}(level : integer, walkparams : S1TTWParams, skl : bits(2),
                                  ia : bits(64), tablebase : FullAddress,
                                  descriptor : bits(N)) => FullAddress
begin
    // Input Address size
    let iasize : AddressSize = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{} = granulebits - descsize2;
    let levels : integer = FINAL_LEVEL - level;

    var index : bits(NUM_PABITS);

    let lsb : AddressSize = (levels*stride + granulebits) as AddressSize;
    let nstride : integer = if walkparams.d128 == '1' then UInt(skl) + 1 else 1;
    let msb : AddressSize = ((lsb + (stride * nstride)) - 1) as AddressSize;
    index = ZeroExtend{NUM_PABITS}(ia[msb:lsb]::Zeros(descsize2));

    var descaddress : FullAddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S2LeafBase

```
// AArch64_S2LeafBase()
// =====
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

func AArch64_S2LeafBase(N)(descriptor : bits(N), walkparams : S2TTWParams,
    level : integer) => bits(NUM_PABITS)

begin
    var leafbase : bits(NUM_PABITS) = Zeros{};

    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer{} = if walkparams.dl28 == '1' then 4 else 3;
    let stride : integer = granulebits - descsize2;
    let leafsize : integer = granulebits + stride * (FINAL_LEVEL - level);

    leafbase[47:0] = AlignDownP2{48}(descriptor[47:0], leafsize as integer{0..48});

    if walkparams.dl28 == '1' then
        leafbase[55:48] = descriptor[55:48];
    elseif walkparams.tgx == TGx_64KB && (AArch64_PAMax() >= 52 ||
        (ImpDefBool(
            "descriptor[15:12] for 64KB granule are OA[51:48]"))) then
        leafbase[51:48] = descriptor[15:12];
    elseif walkparams.ds == '1' then
        leafbase[51:48] = descriptor[9:8]::descriptor[49:48];
    end;

    return leafbase;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S2SLTTEEntryAddress

```
// AArch64_S2SLTTEEntryAddress()
// =====
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

func AArch64_S2SLTTEEntryAddress(walkparams : S2TTWParams, ipa : bits(NUM_PABITS),
    tablebase : FullAddress) => FullAddress

begin
    let startlevel = AArch64_S2StartLevel(walkparams);
    let iasize = AArch64_IASize(walkparams.txsz);
    let granulebits = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer{} = if walkparams.dl28 == '1' then 4 else 3;
    let stride : integer{} = granulebits - descsize2;
    let levels : integer = FINAL_LEVEL - startlevel;

    var index : bits(NUM_PABITS);
    let lsb : AddressSize = (levels*stride + granulebits) as AddressSize;
    let msb : AddressSize = (iasize - 1) as AddressSize;
    index = ZeroExtend{NUM_PABITS}(ipa[msb:lsb]::Zeros{descsize2});

    var descaddress : FullAddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S2StartLevel

```
// AArch64_S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

func AArch64_S2StartLevel(walkparams : S2TTWParams) => integer
begin
    if walkparams.d128 == '1' then
        let iasize : AddressSize = AArch64_IASize(walkparams.txsz);
        let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
        let descsize2 : integer{} = 4;
        let stride : integer = granulebits - descsize2;
        var s2startlevel : integer = FINAL_LEVEL - ((iasize-1) - granulebits) DIV stride;
        s2startlevel = s2startlevel + UInt(walkparams.sk1);

        return s2startlevel;
    end;

    case walkparams.tgx of
        when TGx_4KB =>
            case walkparams.sl2::walkparams.sl0 of
                when '000' => return 2;
                when '001' => return 1;
                when '010' => return 0;
                when '011' => return 3;
                when '100' => return -1;
            end;
        when TGx_16KB =>
            case walkparams.sl0 of
                when '00' => return 3;
                when '01' => return 2;
                when '10' => return 1;
                when '11' => return 0;
            end;
        when TGx_64KB =>
            case walkparams.sl0 of
                when '00' => return 3;
                when '01' => return 2;
                when '10' => return 1;
            end;
    end;
end;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S2TTBaseAddress

```
// AArch64_S2TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 2

func AArch64_S2TTBaseAddress(N)(walkparams : S2TTWParams, paspace : PASpace,
                                ttbr : bits(N)) => bits(NUM_PABITS)
begin
    var tablebase : bits(NUM_PABITS) = Zeros{};

    // Input Address size
    let iasize : AddressSize = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{} = granulebits - descsize2;
    let startlevel : integer = AArch64_S2StartLevel(walkparams);
    let levels : integer = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    var tsize : integer = (iasize - (levels*stride + granulebits)) + descsize2;

    if walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if paspace == PAS_Secure then
            tablebase[55:5] = ttbr[55:5];
        else
            tablebase[55:5] = ttbr[87:80]::ttbr[47:5];
        end;
    elsif walkparams.ds == '1' || (walkparams.tgx == TGx_64KB && walkparams.ps == '110' &&
        (IsFeatureImplemented(FEAT_LPA) ||
        ImpDefBool("BADDR expresses 52 bits for 64KB granule"))) then
        tsize = Max(tsize, 6);
        tablebase[51:6] = ttbr[5:2]::ttbr[47:6];
    else
        tablebase[47:1] = ttbr[47:1];
    end;
    tablebase = AlignDownP2{NUM_PABITS}(tablebase, tsize as integer{0..NUM_PABITS});
    return tablebase;
end;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64_S2TTEntryAddress

```
// AArch64_S2TTEntryAddress()
// =====
// Compute translation table descriptor address within the table pointed to by
// the table base

func AArch64_S2TTEntryAddress(level : integer, walkparams : S2TTWParams, skl : bits(2),
                               ipa : bits(NUM_PABITS), tablebase : FullAddress) => FullAddress
begin
    let ipasize : AddressSize = AArch64_IASize(walkparams.txsz);
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer{} = if walkparams.d128 == '1' then 4 else 3;
    let stride : integer{} = granulebits - descsize2;
    let levels : integer = FINAL_LEVEL - level;

    var index : bits(NUM_PABITS);

    let lsb : AddressSize = (levels*stride + granulebits) as AddressSize;
    let nstride : integer = if walkparams.d128 == '1' then UInt(skl) + 1 else 1;
    let msb : AddressSize = ((lsb + (stride * nstride)) - 1) as AddressSize;
    index = ZeroExtend{NUM_PABITS}(ipa[msb:lsb]::Zeros{descsize2});

    var descaddress : FullAddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_AddrTop

```
// AArch64_AddrTop()
// =====
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

func AArch64_AddrTop(tbid : bit, acctype : AccessType, tbi : bit) => AddressSize
begin
    if tbid == '1' && acctype == AccessType_IFETCH then
        return 63;
    end;

    if tbi == '1' then
        return 55;
    else
        return 63;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_ContiguousBitFaults

```
// AArch64_ContiguousBitFaults()
// =====
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

func AArch64_ContiguousBitFaults(d128 : bit, txsz : bits(6), tgx : TGx, level : integer) => boolean
begin
    // Input Address size
    let iasize : AddressSize = AArch64_IASize(txsz);
    // Translation size
    let tsize : integer = TranslationSize(d128, tgx, level) + ContiguousSize(d128, tgx, level);

    return (tsize > iasize &&
            ImpDefBool("Translation fault on misprogrammed contiguous bit"));
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_IPAIsOutOfRange

```
// AArch64_IPAIsOutOfRange()
// =====
// Check bits not resolved by translation are ZERO

func AArch64_IPAIsOutOfRange(ipa : bits(NUM_PABITS), walkparams : S2TTWParams) => boolean
begin
    //Input Address size
    let iasize : integer{} = AArch64_IASize(walkparams.txsz);

    if iasize < NUM_PABITS then
        return !IsZero(ipa[NUM_PABITS-1:iasize]);
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_OAOutOfRange

```
// AArch64_OAOutOfRange()
// =====
// Returns whether output address is expressed in the configured size number of bits

func AArch64_OAOutOfRange(address : bits(NUM_PABITS), d128 : bit,
                          ds : bit, ps : bits(3), tgx : TGx) => boolean
begin
    // Output Address size
    let oasize : integer{} = AArch64_PhysicalAddressSize(d128, ds, ps, tgx);

    if oasize < NUM_PABITS then
        return !IsZero(address[NUM_PABITS-1:oasize]);
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_PermissionOverlaysApplied

```
// AArch64_PermissionOverlaysApplied()
// =====
// Returns TRUE if Permission overlays are applied for the given access type.

func AArch64_PermissionsOverlaysApplied(acctype : AccessType) => boolean
begin
    case acctype of
        when AccessType_SPE => return FALSE;
        when AccessType_TRBE => return FALSE;
        otherwise => return TRUE;
    end;
end;
```



```

// AArch64_S1CheckPermissions()
// =====
// Checks whether stage 1 access violates permissions of target memory
// and returns a fault record

func AArch64_S1CheckPermissions(fault_in : FaultRecord, va : bits(64), size : integer,
                                regime : Regime, walkstate : TTWState, walkparams : S1TTWParams,
                                accdesc : AccessDescriptor) => FaultRecord

begin
    var fault : FaultRecord = fault_in;
    let permissions : Permissions = walkstate.permissions;
    var slperms : S1AccessControls = AArch64\_S1ComputePermissions(regime, walkstate,
                                                                    walkparams, accdesc);

    if accdesc.acctype == AccessType IFETCH then
        // Flag the access is from a guarded page
        SetInGuardedPage(walkstate.guardedpage == '1' && slperms.x == '1');

        if slperms.overlay && slperms.ox == '0' then
            fault.statuscode = Fault Permission;
            fault.overlay = TRUE;
        elsif (walkstate.memattrs.memtype == MemType Device &&
                ConstrainUnpredictable(Unpredictable INSTRDEVICE) == Constraint FAULT) then
            fault.statuscode = Fault Permission;
        elsif slperms.x == '0' then
            fault.statuscode = Fault Permission;
        end;
    elsif accdesc.acctype == AccessType DC then
        if accdesc.cacheop == CacheOp Invalidate then
            if slperms.overlay && slperms.ow == '0' then
                fault.statuscode = Fault Permission;
                fault.overlay = TRUE;
            elsif slperms.w == '0' then
                fault.statuscode = Fault Permission;
            elsif (walkparams.hd != '1' && walkparams.pie == '1' &&
                    permissions.ndirty == '1') then
                fault.statuscode = Fault Permission;
                fault.dirtybit = TRUE;
            end;
        // DC from privileged context which clean cannot generate a Permission fault
        elsif accdesc.el == EL0 then
            if slperms.overlay && slperms.or == '0' then
                fault.statuscode = Fault Permission;
                fault.overlay = TRUE;
            elsif (walkparams.cmw == '1' &&
                    accdesc.cacheop == CacheOp CleanInvalidate &&
                    slperms.overlay && slperms.ow == '0') then
                fault.statuscode = Fault Permission;
                fault.overlay = TRUE;
            elsif slperms.r == '0' then
                fault.statuscode = Fault Permission;
            elsif (walkparams.cmw == '1' &&
                    accdesc.cacheop == CacheOp CleanInvalidate &&
                    slperms.w == '0') then
                fault.statuscode = Fault Permission;
            elsif (walkparams.cmw == '1' && walkparams.hd != '1' && walkparams.pie == '1' &&
                    permissions.ndirty == '1' && accdesc.cacheop == CacheOp CleanInvalidate) then
                fault.statuscode = Fault Permission;
                fault.dirtybit = TRUE;
            end;
        end;
    elsif accdesc.acctype == AccessType IC then
        // IC from privileged context cannot generate Permission fault
        if accdesc.el == EL0 then
            if (slperms.overlay && slperms.or == '0' &&
                ImpDefBool("Permission fault on EL0 IC_IVAU execution")) then
                fault.statuscode = Fault Permission;
                fault.overlay = TRUE;
            elsif walkparams.cmw == '1' && slperms.overlay && slperms.ow == '0' then
                fault.statuscode = Fault Permission;
                fault.overlay = TRUE;
            elsif (slperms.r == '0' &&
                ImpDefBool("Permission fault on EL0 IC_IVAU execution")) then
                fault.statuscode = Fault Permission;
            elsif walkparams.cmw == '1' && slperms.w == '0' then
                fault.statuscode = Fault Permission;
            elsif (walkparams.cmw == '1' && walkparams.hd != '1' && walkparams.pie == '1' &&
                    permissions.ndirty == '1') then
                fault.statuscode = Fault Permission;
                fault.dirtybit = TRUE;
            end;
        end;
    elsif IsFeatureImplemented(FEAT_GCS) && accdesc.acctype == AccessType GCS then
        if slperms.gcs == '0' then
            fault.statuscode = Fault Permission;
        elsif accdesc.write && walkparams.hd != '1' && permissions.ndirty == '1' then
            fault.statuscode = Fault Permission;
            fault.dirtybit = TRUE;
            fault.write = TRUE;
        end;
    end;
end

```

```

    end;
elseif accdesc.read && slperms.overlay && slperms.or == '0' then
    fault.statuscode = Fault\_Permission;
    fault.overlay    = TRUE;
    fault.write      = FALSE;
elseif accdesc.write && slperms.overlay && slperms.ow == '0' then
    fault.statuscode = Fault\_Permission;
    fault.overlay    = TRUE;
    fault.write      = TRUE;
elseif accdesc.read && slperms.r == '0' then
    fault.statuscode = Fault\_Permission;
    fault.write      = FALSE;
elseif accdesc.write && slperms.w == '0' then
    fault.statuscode = Fault\_Permission;
    fault.write      = TRUE;
elseif (accdesc.write && accdesc.tagaccess &&
        walkstate.memattrs.tags == MemTag\_CanonicalyTagged) then
    fault.statuscode = Fault\_Permission;
    fault.write      = TRUE;
    fault.sltagnotdata = TRUE;
elseif (accdesc.write && walkparams.hd != '1' && walkparams.pie == '1' &&
        permissions.ndirty == '1') then
    fault.statuscode = Fault\_Permission;
    fault.dirtybit   = TRUE;
    fault.write      = TRUE;
end;

return fault;
end;

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S1ComputePermissions

```

// AArch64_S1ComputePermissions()
// =====
// Computes the overall stage 1 permissions

func AArch64_S1ComputePermissions(regime : Regime, walkstate : TTWState,
                                   walkparams : S1TTWParams,
                                   accdesc : AccessDescriptor) => S1AccessControls
begin
    let permissions : Permissions = walkstate.permissions;
    var slperms : S1AccessControls;

    if walkparams.pie == '1' then
        slperms = AArch64\_S1IndirectBasePermissions(regime, walkstate, walkparams, accdesc);
    else
        slperms = AArch64\_S1DirectBasePermissions(regime, walkstate, walkparams, accdesc);
    end;

    if accdesc.el == EL0 && !AArch64\_S1E0POEnabled(regime, walkparams.nv1) then
        slperms.overlay = FALSE;
    elseif accdesc.el != EL0 && !AArch64\_S1POEnabled(regime) then
        slperms.overlay = FALSE;
    end;

    if slperms.overlay then
        let sloverlay_perms : S1AccessControls = AArch64\_S1OverlayPermissions(regime,
                                                                              walkstate,
                                                                              accdesc);

        slperms.or = sloverlay_perms.or;
        slperms.ow = sloverlay_perms.ow;
        slperms.ox = sloverlay_perms.ox;
    end;

    if slperms.overlay then
        // If WXN and the overlay X permission is present, the overlay W permission is removed.
        slperms.ow = slperms.ow AND NOT(slperms.wxn AND slperms.ox);
    else
        // If WXN and the W and X permissions are present, the X permission is removed.
        // The W and X permissions are factored into the WXN computation.
        slperms.x = slperms.x AND NOT(slperms.wxn);
    end;

    return slperms;
end;

```



```

// AArch64_S1DirectBasePermissions()
// =====
// Computes the stage 1 direct base permissions

func AArch64_S1DirectBasePermissions(regime : Regime, walkstate : TTWState,
                                     walkparams : S1TTWParams,
                                     accdesc : AccessDescriptor) => S1AccessControls
begin
    var r, w, x, wxn : bit;
    var pr, pw, px, pwxn : bit;
    var ur, uw, ux, uwxn : bit;
    var permissions : Permissions = walkstate.permissions;
    var slperms : S1AccessControls;
    // Descriptors marked with DBM set have the effective value of AP[2] cleared.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if permissions.dbm == '1' && walkparams.hd == '1' then
        permissions.ap[2] = '0';
    end;

    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case permissions.ap[2:1] of
            when '00' => (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
            when '01' => (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
            when '10' => (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
            when '11' => (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only
        end;

        // Apply hierarchical permissions
        case permissions.ap_table of
            when '00' => (pr,pw,ur,uw) = (pr, pw, ur, uw); // No effect
            when '01' => (pr,pw,ur,uw) = (pr, pw, '0','0'); // Privileged access
            when '10' => (pr,pw,ur,uw) = (pr, '0', ur, '0'); // Read-only
            when '11' => (pr,pw,ur,uw) = (pr, '0','0','0'); // Read-only, privileged access
        end;

        // Locations writable by unprivileged cannot be executed by privileged
        px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
        ux = NOT(permissions.uxn OR permissions.uxn_table);
        pwxn = walkparams.wxn AND pw AND px;
        uwxn = walkparams.wxn AND uw AND ux;

        if (IsFeatureImplemented(FEAT_PAN) && accdesc.pan && !(regime == Regime\_EL10 &&
            walkparams.nv1 == '1')) then
            if (ImpDefBool("SCR_EL3.SIF affects EPAN") && accdesc.ss == SS\_Secure &&
                walkstate.baseaddress.paspace == PAS\_NonSecure && walkparams.sif == '1') then
                ux = '0';
            end;

            if (ImpDefBool("Realm EL2&0 regime affects EPAN") && accdesc.ss == SS\_Realm &&
                regime == Regime\_EL20 && walkstate.baseaddress.paspace != PAS\_Realm) then
                ux = '0';
            end;

            let pan : bit = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);
        end;
    else
        // Apply leaf permissions
        case permissions.ap[2] of
            when '0' => (pr,pw) = ('1','1'); // No effect
            when '1' => (pr,pw) = ('1','0'); // Read-only
        end;

        // Apply hierarchical permissions
        case permissions.ap_table[1] of
            when '0' => (pr,pw) = (pr, pw); // No effect
            when '1' => (pr,pw) = (pr, '0'); // Read-only
        end;

        px = NOT(permissions.xn OR permissions.xn_table);
        pwxn = walkparams.wxn AND pw AND px;
    end;

    (r,w,x,wxn) = if accdesc.el == EL0 then (ur,uw,ux,uwxn) else (pr,pw,px,pwxn);

    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if accdesc.ss == SS\_Secure && walkstate.baseaddress.paspace == PAS\_NonSecure then
        x = x AND NOT(walkparams.sif);
    end;
    // Prevent execution from non-Root space by Root
    if accdesc.ss == SS\_Root && walkstate.baseaddress.paspace != PAS\_Root then
        x = '0';
    end;
    // Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
    if (accdesc.ss == SS\_Realm && regime IN {Regime\_EL2, Regime\_EL20} &&

```

```

        walkstate.baseaddress.paspace != PAS Realm) then
    x = '0';
end;

slperms.r    = r;
slperms.w    = w;
slperms.x    = x;
slperms.gcs  = '0';
slperms.wxn  = wxn;
slperms.overlay = TRUE;

return slperms;
end;

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S1HasAlignmentFaultDueToMemType

```

// AArch64_S1HasAlignmentFaultDueToMemType()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses due to memory type

func AArch64_S1HasAlignmentFaultDueToMemType(regime : Regime, accdesc : AccessDescriptor,
        aligned : boolean, ntlsmid : bit,
        memattrs : MemoryAttributes) => boolean

begin
    if accdesc.exclusive || accdesc.atomicop || accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        if (!aligned && !(IsWBShareable(memattrs) && AArch64\_S1DCacheEnabled(regime)) &&
            ConstrainUnpredictableBool(Unpredictable\_LSE2\_ALIGNMENT\_FAULT)) then
            return TRUE;
        end;
    end;

    if memattrs.memtype != MemType\_Device then
        return FALSE;
    elsif ((accdesc.acctype == AccessType\_DCZero && accdesc.cachetype == CacheType\_Tag) ||
        accdesc.stzgm) then
        return ConstrainUnpredictable(Unpredictable\_DEVICE\_TAGSTORE) == Constraint\_FAULT;
    elsif accdesc.a32lsmid && ntlsmid == '0' then
        return memattrs.device != DeviceType\_GRE;
    elsif accdesc.acctype == AccessType\_DCZero then
        return TRUE;
    elsif !aligned then
        return !(ImpDefBool("Device location supports unaligned access"));
    else
        return FALSE;
    end;
end;

```



```

// AArch64_S1IndirectBasePermissions()
// =====
// Computes the stage 1 indirect base permissions

func AArch64_S1IndirectBasePermissions(regime : Regime, walkstate : TTWState,
                                       walkparams : S1TTWParams,
                                       accdesc : AccessDescriptor) => S1AccessControls
begin

    var r, w, x, gcs, wxn, overlay : bit;
    var pr, pw, px, pgcs, pwxn, p_overlay : bit;
    var ur, uw, ux, ugcs, uwxn, u_overlay : bit;
    let permissions : Permissions = walkstate.permissions;
    var slperms : S1AccessControls;

    // Apply privileged indirect permissions
    case permissions.ppi of
        when '0000' => (pr,pw,px,pgcs) = ('0','0','0','0'); // No access
        when '0001' => (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
        when '0010' => (pr,pw,px,pgcs) = ('0','0','1','0'); // Privileged execute
        when '0011' => (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
        when '0100' => (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '0101' => (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
        when '0110' => (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '0111' => (pr,pw,px,pgcs) = ('1','1','1','1'); // Privileged read, write and execute
        when '1000' => (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
        when '1001' => (pr,pw,px,pgcs) = ('1','0','0','1'); // Privileged read and gcs
        when '1010' => (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
        when '1011' => (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '1100' => (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
        when '1101' => (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '1110' => (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '1111' => (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
    end;

    p_overlay = NOT(permissions.ppi[3]);
    pwxn = if permissions.ppi == '0110' then '1' else '0';
    var reserved_upi : boolean = FALSE;

    if HasUnprivileged(regime) then
        // Apply unprivileged indirect permissions
        case permissions.upi of
            when '0000' => (ur,uw,ux,ugcs) = ('0','0','0','0'); // No access
            when '0001' => (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged read
            when '0010' => (ur,uw,ux,ugcs) = ('0','0','1','0'); // Unprivileged execute
            when '0011' => (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged read and execute
            when '0100' =>
                (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
                reserved_upi = TRUE;
            when '0101' => (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged read and write
            when '0110' => (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read, write and execute
            when '0111' => (ur,uw,ux,ugcs) = ('1','1','1','1'); // Unprivileged read, write and execute
            when '1000' => (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged read
            when '1001' => (ur,uw,ux,ugcs) = ('1','0','0','1'); // Unprivileged read and gcs
            when '1010' => (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged read and execute
            when '1011' =>
                (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
                reserved_upi = TRUE;
            when '1100' => (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged read and write
            when '1101' =>
                (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
                reserved_upi = TRUE;
            when '1110' => (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read, write
                // and execute
            when '1111' =>
                (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
                reserved_upi = TRUE;
        end;

        u_overlay = NOT(permissions.upi[3]);
        uwxn = if permissions.upi == '0110' then '1' else '0';

        // If the decoded permissions has either px or pgcs along with either uw or ugcs,
        // then all effective Stage 1 Base Permissions are set to 0
        if ((px == '1' || pgcs == '1') && (uw == '1' || ugcs == '1')) then
            (pr,pw,px,pgcs) = ('0','0','0','0');
            (ur,uw,ux,ugcs) = ('0','0','0','0');
        end;
    end;

    // Prevent execution from Non-secure space by PE in secure state
    if accdesc.ss == SS\_Secure && walkstate.baseaddress.paspace == PAS\_NonSecure then
        pgcs = '0';
        if HasUnprivileged(regime) then ugcs = '0'; end;

        if walkparams.sif == '1' then
            px = '0';
            if HasUnprivileged(regime) then ux = '0'; end;

```

```

    end;
end;

// Prevent execution from non-Root space by Root
if accdesc.ss == SS\_Root && walkstate.baseaddress.paspace != PAS\_Root then
    (px, pgcs) = ('0','0');
    if HasUnprivileged(regime) then
        (ux, ugcs) = ('0','0');
    end;
end;

// Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
if (accdesc.ss == SS\_Realm && regime IN {Regime\_EL2, Regime\_EL20} &&
    walkstate.baseaddress.paspace != PAS\_Realm) then
    (px, pgcs) = ('0','0');
    if HasUnprivileged(regime) then
        (ux, ugcs) = ('0','0');
    end;
end;

if (HasUnprivileged(regime) && IsFeatureImplemented(FEAT_PAN) && accdesc.pan &&
    !(regime == Regime\_EL10 && walkparams.nv1 == '1') &&
    PSTATE.PAN == '1' && !IsZero(permissions.upi)) then
    if reserved_upi then
        if ImpDefBool("PAN applies when S1UnprivBasePerm is reserved") then
            (pr,pw) = ('0','0');
        end;
    elseif (!IsZero(ur:uw:ux:ugcs) ||
        ImpDefBool("PAN applies when S1UnprivBasePerms are removed")) then
        (pr,pw) = ('0','0');
    end;
end;

if accdesc.el == EL0 then
    (r,w,x,gcs,wxn,overlay) = (ur,uw,ux,ugcs,uwxn,u_overlay);
else
    (r,w,x,gcs,wxn,overlay) = (pr,pw,px,pgcs,pwxn,p_overlay);
end;

slperms.r      = r;
slperms.w      = w;
slperms.x      = x;
slperms.gcs    = gcs;
slperms.wxn    = wxn;
slperms.overlay = overlay == '1';

return slperms;
end;

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S1OAOOutOfRange

```

// AArch64_S1OAOOutOfRange()
// =====
// Returns whether stage 1 output address is expressed in the configured size number of bits

func AArch64_S1OAOOutOfRange(address : bits(NUM\_PABITS), walkparams : S1TTWParams) => boolean
begin
    return AArch64\_OAOOutOfRange(address, walkparams.dl28, walkparams.ds, walkparams.ps,
        walkparams.tgx);
end;

```


Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S1OverlayPermissions

```
// AArch64_S1OverlayPermissions()
// =====
// Computes the stage 1 overlay permissions

func AArch64_S1OverlayPermissions(regime : Regime, walkstate : TTWState,
                                   accdesc : AccessDescriptor) => S1AccessControls
begin
    var r, w, x : bit;
    var pr, pw, px : bit;
    var ur, uw, ux : bit;
    let permissions : Permissions = walkstate.permissions;
    var sloverlay_perms : S1AccessControls;

    let por : S1PORType = AArch64_S1POR(regime);
    let bit_index : integer{} = 4 * UInt(permissions.po_index);

    let ppo : bits(4) = por[bit_index+3:bit_index];

    // Apply privileged overlay permissions
    case ppo of
        when '0000' => (pr,pw,px) = ('0','0','0'); // No access
        when '0001' => (pr,pw,px) = ('1','0','0'); // Privileged read
        when '0010' => (pr,pw,px) = ('0','0','1'); // Privileged execute
        when '0011' => (pr,pw,px) = ('1','0','1'); // Privileged read and execute
        when '0100' => (pr,pw,px) = ('0','1','0'); // Privileged write
        when '0101' => (pr,pw,px) = ('1','1','0'); // Privileged read and write
        when '0110' => (pr,pw,px) = ('0','1','1'); // Privileged write and execute
        when '0111' => (pr,pw,px) = ('1','1','1'); // Privileged read, write and execute
        when '1xxx' => (pr,pw,px) = ('0','0','0'); // Reserved
    end;

    if HasUnprivileged(regime) then
        var upo : bits(4) = '0000';
        if ! (HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
            ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            upo = POR_EL0()[bit_index+3:bit_index];
        end;

        // Apply unprivileged overlay permissions
        case upo of
            when '0000' => (ur,uw,ux) = ('0','0','0'); // No access
            when '0001' => (ur,uw,ux) = ('1','0','0'); // Unprivileged read
            when '0010' => (ur,uw,ux) = ('0','0','1'); // Unprivileged execute
            when '0011' => (ur,uw,ux) = ('1','0','1'); // Unprivileged read and execute
            when '0100' => (ur,uw,ux) = ('0','1','0'); // Unprivileged write
            when '0101' => (ur,uw,ux) = ('1','1','0'); // Unprivileged read and write
            when '0110' => (ur,uw,ux) = ('0','1','1'); // Unprivileged write and execute
            when '0111' => (ur,uw,ux) = ('1','1','1'); // Unprivileged read, write and execute
            when '1xxx' => (ur,uw,ux) = ('0','0','0'); // Reserved
        end;
    end;

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    sloverlay_perms.or = r;
    sloverlay_perms.ow = w;
    sloverlay_perms.ox = x;

    return sloverlay_perms;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S1TxSZFaults

```
// AArch64_S1TxSZFaults()
// =====
// Detect whether configuration of stage 1 TxSZ field generates a fault

func AArch64_S1TxSZFaults(regime : Regime, walkparams : S1TTWParams) => boolean
begin
    let mintxs : integer = AArch64_S1MinTxSZ(regime, walkparams);
    let maxtxs : integer = AArch64_MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxs then
        return (IsFeatureImplemented(FEAT_LVA) ||
            ImpDefBool("Fault on TxSZ value below minimum"));
    end;
    if UInt(walkparams.txsz) > maxtxs then
        return ImpDefBool("Fault on TxSZ value above maximum");
    end;

    return FALSE;
end;
```



```

// AArch64_S2CheckPermissions()
// =====
// Verifies memory access with available permissions.

func AArch64_S2CheckPermissions(fault_in : FaultRecord, walkstate : TTWState,
                                walkparams : S2TTWParams, ipa : AddressDescriptor,
                                accdesc : AccessDescriptor) => (FaultRecord, boolean)

begin
    let memtype : MemType = walkstate.memattrs.memtype;
    let permissions : Permissions = walkstate.permissions;
    var fault : FaultRecord = fault_in;
    let s2perms : S2AccessControls = AArch64\_S2ComputePermissions(permissions, walkparams,
                                                                    accdesc);

    var r, w : bit;
    var or, ow : bit;

    if accdesc.acctype == AccessType TTW then
        r = s2perms.r_mmu;
        w = s2perms.w_mmu;
        or = s2perms.or_mmu;
        ow = s2perms.ow_mmu;
    elseif accdesc.rcw then
        r = s2perms.r_rcw;
        w = s2perms.w_rcw;
        or = s2perms.or_rcw;
        ow = s2perms.ow_rcw;
    else
        r = s2perms.r;
        w = s2perms.w;
        or = s2perms.or;
        ow = s2perms.ow;
    end;

    if accdesc.acctype == AccessType TTW then
        if (accdesc.toplevel && accdesc.varange == VRange LOWER &&
            ((walkparams.tl0 == '1' && s2perms.toplevel0 == '0') ||
             (walkparams.tl1 == '1' && s2perms.[toplevel1,toplevel0] == '10'))) then
            fault.statuscode = Fault Permission;
            fault.toplevel = TRUE;
        elseif (accdesc.toplevel && accdesc.varange == VRange UPPER &&
            ((walkparams.tl1 == '1' && s2perms.toplevel1 == '0') ||
             (walkparams.tl0 == '1' && s2perms.[toplevel1,toplevel0] == '01'))) then
            fault.statuscode = Fault Permission;
            fault.toplevel = TRUE;
        // Stage 2 Permission fault due to AssuredOnly check
        elseif (walkstate.s2assuredonly == '1' && !ipa.slassured) then
            fault.statuscode = Fault Permission;
            fault.assuredonly = TRUE;

        elseif s2perms.overlay && or == '0' then
            fault.statuscode = Fault Permission;
            fault.overlay = TRUE;
        elseif accdesc.write && s2perms.overlay && ow == '0' then
            fault.statuscode = Fault Permission;
            fault.overlay = TRUE;

        elseif walkparams.ptw == '1' && memtype == MemType Device then
            fault.statuscode = Fault Permission;
        // Prevent translation table walks in Non-secure space by Realm state
        elseif accdesc.ss == SS Realm && walkstate.baseaddress.paspace != PAS Realm then
            fault.statuscode = Fault Permission;
        elseif r == '0' then
            fault.statuscode = Fault Permission;
        elseif accdesc.write && w == '0' then
            fault.statuscode = Fault Permission;
            fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS() && permissions.dbm == '1';
        elseif (accdesc.write && walkparams.s2pie == '1' && permissions.s2dirty == '0' &&
            (walkparams.hd != '1' ||
             (accdesc.acctype != AccessType AT &&
              walkparams.hdbss == '1' && !CanAppendToHDBSS()))) then
            fault.statuscode = Fault Permission;
            fault.dirtybit = TRUE;
            fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS();
        end;

    // Stage 2 Permission fault due to AssuredOnly check
    elseif ((walkstate.s2assuredonly == '1' && !ipa.slassured) ||
            (walkstate.s2assuredonly != '1' && IsFeatureImplemented(FEAT_GCS) &&
             VTCR_EL2().GCSH == '1' && accdesc.acctype == AccessType GCS && accdesc.el != EL0)) then
        fault.statuscode = Fault Permission;
        fault.assuredonly = TRUE;

    elseif accdesc.acctype == AccessType IFETCH then
        if s2perms.overlay && s2perms.ox == '0' then
            fault.statuscode = Fault Permission;
            fault.overlay = TRUE;
        elseif (memtype == MemType Device &&

```

```

        ConstrainUnpredictable(Unpredictable INSTRDEVICE) == Constraint FAULT) then
        fault.statuscode = Fault Permission;

// Prevent execution from Non-secure space by Realm state
elseif accdesc.ss == SS Realm && walkstate.baseaddress.paspace != PAS Realm then
    fault.statuscode = Fault Permission;
elseif s2perms.x == '0' then
    fault.statuscode = Fault Permission;
end;

elseif accdesc.acctype == AccessType DC then
    if accdesc.cacheop == CacheOp Invalidate then
        if !ELUsingAArch32(EL1) && s2perms.overlay && ow == '0' then
            fault.statuscode = Fault Permission;
            fault.overlay = TRUE;
        elseif !ELUsingAArch32(EL1) && w == '0' then
            fault.statuscode = Fault Permission;
        elseif (walkparams.hd != '1' && walkparams.s2pie == '1' &&
            permissions.s2dirty == '0') then
            fault.statuscode = Fault Permission;
            fault.dirtybit = TRUE;
        end;
    elseif !ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' then
        fault.statuscode = Fault Permission;
        fault.overlay = TRUE;
    elseif (walkparams.cmw == '1' && accdesc.cacheop == CacheOp CleanInvalidate &&
        s2perms.overlay && ow == '0') then
        fault.statuscode = Fault Permission;
        fault.overlay = TRUE;
    elseif !ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' then
        fault.statuscode = Fault Permission;
    elseif (walkparams.cmw == '1' && accdesc.cacheop == CacheOp CleanInvalidate &&
        w == '0') then
        fault.statuscode = Fault Permission;
    elseif (walkparams.cmw == '1' && accdesc.cacheop == CacheOp CleanInvalidate &&
        walkparams.hd != '1' && walkparams.s2pie == '1' && permissions.s2dirty == '0') then
        fault.statuscode = Fault Permission;
        fault.dirtybit = TRUE;
    end;

elseif accdesc.acctype == AccessType IC then
    if (!ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' &&
        ImpDefBool("Permission fault on EL0 IC IVAU execution")) then
        fault.statuscode = Fault Permission;
        fault.overlay = TRUE;
    elseif walkparams.cmw == '1' && s2perms.overlay && ow == '0' then
        fault.statuscode = Fault Permission;
        fault.overlay = TRUE;
    elseif (!ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' &&
        ImpDefBool("Permission fault on EL0 IC IVAU execution")) then
        fault.statuscode = Fault Permission;
    elseif walkparams.cmw == '1' && w == '0' then
        fault.statuscode = Fault Permission;
    elseif (walkparams.cmw == '1' && walkparams.hd != '1' && walkparams.s2pie == '1' &&
        permissions.s2dirty == '0') then
        fault.statuscode = Fault Permission;
        fault.dirtybit = TRUE;
    end;

elseif accdesc.read && s2perms.overlay && or == '0' then
    fault.statuscode = Fault Permission;
    fault.overlay = TRUE;
    fault.write = FALSE;
elseif accdesc.write && s2perms.overlay && ow == '0' then
    fault.statuscode = Fault Permission;
    fault.overlay = TRUE;
    fault.write = TRUE;
elseif accdesc.read && r == '0' then
    fault.statuscode = Fault Permission;
    fault.write = FALSE;
elseif accdesc.write && w == '0' then
    fault.statuscode = Fault Permission;
    fault.write = TRUE;
    fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS() && permissions.dbm == '1';
elseif (IsFeatureImplemented(FEAT_MTE_PERM) &&
    (accdesc.tagchecked &&
        AArch64 EffectiveTCF(accdesc.el, accdesc.read) != TCFTType Ignore) ||
        accdesc.tagaccess) &&
    ipa.memattr.tags == MemTag AllocationTagged &&
    permissions.s2tag_na == '1' && S2DCacheEnabled()) then
    fault.statuscode = Fault Permission;
    fault.tagaccess = TRUE;
    fault.write = accdesc.tagaccess && accdesc.write;
elseif (accdesc.write &&
    (walkparams.hd != '1' || (walkparams.hdbss == '1' && !CanAppendToHDBSS())) &&
    walkparams.s2pie == '1' && permissions.s2dirty == '0') then
    fault.statuscode = Fault Permission;
    fault.dirtybit = TRUE;
    fault.write = TRUE;

```

```

        fault.hdbssf      = walkparams.hdbss == '1' && !CanAppendToHDBSS();
    end;
    // MRO* allows only RCW and MMU writes
    var mro : boolean;
    if s2perms.overlay then
        mro = (s2perms.[w,w_rcw,w_mmu] AND s2perms.[ow,ow_rcw,ow_mmu]) == '011';
    else
        mro = s2perms.[w,w_rcw,w_mmu] == '011';
    end;

    return (fault, mro);
end;

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S2ComputePermissions

```

// AArch64_S2ComputePermissions()
// =====
// Compute the overall stage 2 permissions.

func AArch64_S2ComputePermissions(permissions : Permissions, walkparams : S2TTWParams,
                                   accdesc : AccessDescriptor) => S2AccessControls
begin
    var s2perms : S2AccessControls;

    if walkparams.s2pie == '1' then
        s2perms = AArch64\_S2IndirectBasePermissions(permissions, accdesc);
        s2perms.overlay = IsFeatureImplemented(FEAT_S2POE) && VTCR_EL2().S2POE == '1';
        if s2perms.overlay then
            let s2overlay_perms : S2AccessControls = AArch64\_S2OverlayPermissions(permissions, accdesc);

            s2perms.or      = s2overlay_perms.or;
            s2perms.ow      = s2overlay_perms.ow;
            s2perms.ox      = s2overlay_perms.ox;
            s2perms.or_rcw  = s2overlay_perms.or_rcw;
            s2perms.ow_rcw  = s2overlay_perms.ow_rcw;
            s2perms.or_mmu  = s2overlay_perms.or_mmu;
            s2perms.ow_mmu  = s2overlay_perms.ow_mmu;

            // Toplevel is applicable only when the effective S2 permissions is MRO
            if ((s2perms.[w,w_rcw,w_mmu] AND s2perms.[ow,ow_rcw,ow_mmu]) == '011') then
                s2perms.toplevel0 = s2perms.toplevel0 OR s2overlay_perms.toplevel0;
                s2perms.toplevel1 = s2perms.toplevel1 OR s2overlay_perms.toplevel1;
            else
                s2perms.toplevel0 = '0';
                s2perms.toplevel1 = '0';
            end;
        end;
    else
        s2perms = AArch64\_S2DirectBasePermissions(permissions, accdesc, walkparams);
    end;

    return s2perms;
end;

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S2DirectBasePermissions

```
// AArch64_S2DirectBasePermissions()
// =====
// Computes the stage 2 direct base permissions.

func AArch64_S2DirectBasePermissions(permissions : Permissions,
                                     accdesc : AccessDescriptor,
                                     walkparams : S2TWPParams) => S2AccessControls

begin
    var s2perms : S2AccessControls;
    var w : bit;
    let r : bit = permissions.s2ap[0];
    if permissions.s2ap[1] == '1' then
        w = '1';
    // Descriptors marked with DBM set have the effective value of S2AP[1] set.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    elsif permissions.dbm == '1' && walkparams.hd == '1' then
        // An update occurs here, conditional to being able to append to HDBSS
        if walkparams.hdbss == '1' then
            w = if CanAppendToHDBSS() then '1' else '0';
        else
            w = '1';
        end;
    else
        w = '0';
    end;

    var px, ux : bit;
    case (permissions.s2xn::permissions.s2xnx) of
        when '00' => (px,ux) = ('1','1');
        when '01' => (px,ux) = ('0','1');
        when '10' => (px,ux) = ('0','0');
        when '11' => (px,ux) = ('1','0');
    end;

    let x : bit = if accdesc.el == ELO then ux else px;
    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r;
    s2perms.w_rcw = w;
    s2perms.r_mmu = r;
    s2perms.w_mmu = w;
    s2perms.toplevel0 = '0';
    s2perms.toplevel1 = '0';
    s2perms.overlay = FALSE;

    return s2perms;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S2HasAlignmentFaultDueToMemType

```
// AArch64_S2HasAlignmentFaultDueToMemType()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses due to memory type

func AArch64_S2HasAlignmentFaultDueToMemType(accdesc : AccessDescriptor, aligned : boolean,
                                              memattrs : MemoryAttributes) => boolean

begin
    if accdesc.exclusive || accdesc.atomicop || accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        if (!aligned && !(IsWBShareable(memattrs) && S2DCacheEnabled()) &&
            ConstrainUnpredictableBool(Unpredictable LSE2 ALIGNMENT FAULT)) then
            return TRUE;
        end;
    end;

    if memattrs.memtype != MemType\_Device then
        return FALSE;
    elsif ((accdesc.acctype == AccessType\_DCZero && accdesc.cachetype == CacheType\_Tag) ||
        accdesc.stzgm) then
        return ConstrainUnpredictable(Unpredictable DEVICETAGSTORE) == Constraint\_FAULT;
    elsif accdesc.acctype == AccessType\_DCZero then
        return TRUE;
    elsif !aligned then
        return !(ImpDefBool("Device location supports unaligned access"));
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S2InconsistentSL

```
// AArch64_S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

func AArch64_S2InconsistentSL(walkparams : S2TTWParams) => boolean
begin
    let startlevel : integer      = AArch64_S2StartLevel(walkparams);
    let levels : integer          = FINAL_LEVEL - startlevel;
    let granulebits : AddressSize = TGxGranuleBits(walkparams.tgx);
    let descsize2 : integer{}    = 3;
    let stride : integer{}       = granulebits - descsize2;

    // Input address size must at least be large enough to be resolved from the start level
    let sl_min_iasize : integer = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accommodate 1 more stride in the level + concatenation of up to 2^4 tables
    let sl_max_iasize : integer = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    let iasize : AddressSize    = AArch64_IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S2IndirectBasePermissions

```
// AArch64_S2IndirectBasePermissions()
// =====
// Computes the stage 2 indirect base permissions.

func AArch64_S2IndirectBasePermissions(permissions : Permissions,
                                       accdesc : AccessDescriptor) => S2AccessControls
begin
    var r, w : bit;
    var r_rcw, w_rcw : bit;
    var r_mmu, w_mmu : bit;
    var px, ux : bit;
    var toplevel0, toplevel1 : bit;
    var s2perms : S2AccessControls;

    let s2pi : bits(4) = permissions.s2pi;
    case s2pi of
        when '0000' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // No Access
        when '0001' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // Reserved
        when '0010' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1'); // MRO
        when '0011' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1'); // MRO-TL1
        when '0100' => (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0'); // Write Only
        when '0101' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // Reserved
        when '0110' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1'); // MRO-TL0
        when '0111' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1'); // MRO-TL01
        when '1000' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0'); // Read Only
        when '1001' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0'); // Read, Unpriv Execute
        when '1010' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0'); // Read, Priv Execute
        when '1011' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0'); // Read, All Execute
        when '1100' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','0','1'); // RW
        when '1101' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','0','1'); // RW, Unpriv Execute
        when '1110' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','0','1'); // RW, Priv Execute
        when '1111' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','0','1'); // RW, All Execute
    end;

    let x : bit = if accdesc.el == ELO then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2pi of
        when '0110' => (toplevel0,toplevel1) = ('1','0');
        when '0011' => (toplevel0,toplevel1) = ('0','1');
        when '0111' => (toplevel0,toplevel1) = ('1','1');
        otherwise => (toplevel0,toplevel1) = ('0','0');
    end;

    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r_rcw;
    s2perms.r_mmu = r_mmu;
    s2perms.w_rcw = w_rcw;
    s2perms.w_mmu = w_mmu;
    s2perms.toplevel0 = toplevel0;
    s2perms.toplevel1 = toplevel1;

    return s2perms;
end;
```


Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S2InvalidSL

```
// AArch64_S2InvalidSL()
// =====
// Detect invalid configuration of SL field

func AArch64_S2InvalidSL(walkparams : S2TTWParams) => boolean
begin
  case walkparams.tgx of
    when TGx_4KB =>
      case walkparams.sl2::walkparams.sl0 of
        when '1x1' => return TRUE;
        when '11x' => return TRUE;
        when '100' => return AArch64_PAMax() < 52;
        when '010' => return AArch64_PAMax() < 44;
        when '011' => return !IsFeatureImplemented(FEAT_TTST);
        otherwise => return FALSE;
      end;
    when TGx_16KB =>
      case walkparams.sl0 of
        when '11' => return walkparams.ds == '0' || AArch64_PAMax() < 52;
        when '10' => return AArch64_PAMax() < 42;
        otherwise => return FALSE;
      end;
    when TGx_64KB =>
      case walkparams.sl0 of
        when '11' => return TRUE;
        when '10' => return AArch64_PAMax() < 44;
        otherwise => return FALSE;
      end;
  end;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S2OAOOutOfRange

```
// AArch64_S2OAOOutOfRange()
// =====
// Returns whether stage 2 output address is expressed in the configured size number of bits

func AArch64_S2OAOOutOfRange(address : bits(NUM_PABITS), walkparams : S2TTWParams) => boolean
begin
  return AArch64_OAOOutOfRange(address, walkparams.dl28, walkparams.ds, walkparams.ps,
                                walkparams.tgx);
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S2OverlayPermissions

```
// AArch64_S2OverlayPermissions()
// =====
// Computes the stage 2 overlay permissions.

func AArch64_S2OverlayPermissions(permissions : Permissions,
                                   accdesc : AccessDescriptor) => S2AccessControls
begin
    var r, w : bit;
    var r_rcw, w_rcw : bit;
    var r_mmu, w_mmu : bit;
    var px, ux : bit;
    var toplevel0, toplevel1 : bit;
    var s2overlay_perms : S2AccessControls;

    let index : integer{} = 4 * UInt(permissions.s2po_index);
    var s2po : bits(4) = '0000';
    if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
        ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
        s2po = S2POR_EL1()[index+3:index];
    end;
    case s2po of
        when '0000' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // No Access
        when '0001' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // Reserved
        when '0010' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO
        when '0011' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO-TL1
        when '0100' => (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0'); // Write Only
        when '0101' => (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // Reserved
        when '0110' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO-TL0
        when '0111' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO-TL01
        when '1000' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0'); // Read Only
        when '1001' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0'); // Read, Unpriv Execute
        when '1010' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0'); // Read, Priv Execute
        when '1011' => (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0'); // Read, All Execute
        when '1100' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1'); // RW
        when '1101' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1'); // RW, Unpriv Execute
        when '1110' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1'); // RW, Priv Execute
        when '1111' => (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1'); // RW, All Execute
    end;

    let x : bit = if accdesc.el == EL0 then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2po of
        when '0110' => (toplevel0,toplevel1) = ('1','0');
        when '0011' => (toplevel0,toplevel1) = ('0','1');
        when '0111' => (toplevel0,toplevel1) = ('1','1');
        otherwise => (toplevel0,toplevel1) = ('0','0');
    end;

    s2overlay_perms.or = r;
    s2overlay_perms.ow = w;
    s2overlay_perms.ox = x;
    s2overlay_perms.or_rcw = r_rcw;
    s2overlay_perms.ow_rcw = w_rcw;
    s2overlay_perms.or_mmu = r_mmu;
    s2overlay_perms.ow_mmu = w_mmu;
    s2overlay_perms.toplevel0 = toplevel0;
    s2overlay_perms.toplevel1 = toplevel1;

    return s2overlay_perms;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_S2TxSZFaults

```
// AArch64_S2TxSZFaults()
// =====
// Detect whether configuration of stage 2 TxSZ field generates a fault

func AArch64_S2TxSZFaults(walkparams : S2TTWParams, slaarch64 : boolean) => boolean
begin
    let mintxs : integer = AArch64\_S2MinTxSZ(walkparams, slaarch64);
    let maxtxs : integer = AArch64\_MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txs) < mintxs then
        return (IsFeatureImplemented(FEAT_LPA) ||
            ImpDefBool("Fault on TxSZ value below minimum"));
    end;
    if UInt(walkparams.txs) > maxtxs then
        return ImpDefBool("Fault on TxSZ value above maximum");
    end;

    return FALSE;
end;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64_VAIsOutOfRange

```
// AArch64_VAIsOutOfRange()
// =====
// Check bits not resolved by translation are identical and of accepted value

func AArch64_VAIsOutOfRange(va_in : bits(64), acctype : AccessType,
                             regime : Regime, walkparams : S1TTWParams) => boolean
begin
    var va : bits(64) = va_in;

    let addrtop : AddressSize = AArch64\_AddrTop(walkparams.tbid, acctype, walkparams.tbi);

    // If the VA has a Logical Address Tag then the bits holding the Logical Address Tag are
    // ignored when checking if the address is out of range.
    if walkparams.mtx == '1' && acctype != AccessType\_IFETCH then
        va[59:56] = if AArch64\_GetVARange(va) == VARange\_UPPER then '1111' else '0000';
    end;

    // Input Address size
    let iasize : integer{} = AArch64\_IASize(walkparams.txsz);

    // The min value of TxSZ can be 8, with LVA3 implemented.
    // If TxSZ is set to 8 iasize becomes 64 - 8 = 56
    // If tbi is also set, addrtop becomes 55
    // Then the return statements check va[56:55]
    // The check here is to guard against this corner case.
    if addrtop < iasize then
        return FALSE;
    end;

    if HasUnprivileged(regime) then
        if AArch64\_GetVARange(va) == VARange\_LOWER then
            return !IsZero(va[addrtop:iasize]);
        else
            return !IsOnes(va[addrtop:iasize]);
        end;
    else
        return !IsZero(va[addrtop:iasize]);
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_memattr/AArch64_S2ApplyFWBMemAttrs

```
// AArch64_S2ApplyFWBMemAttrs()
// =====
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

func AArch64_S2ApplyFWBMemAttrs(N)(s1_memattrs : MemoryAttributes, walkparams : S2TTWParams,
                                   descriptor : bits(N)) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;
    let s2_attr : bits(4) = descriptor[5:2];
    let s2_sh : bits(2)   = if walkparams.ds == '1' then walkparams.sh else descriptor[9:8];
    let s2_fnxs : bit     = descriptor[11];

    if s2_attr[2] == '0' then           // S2 Device, S1 any
        let s2_device : DeviceType = DecodeDevice(s2_attr[1:0]);
        memattrs.memtype = MemType\_Device;
        if s1_memattrs.memtype == MemType\_Device then
            memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_device);
        else
            memattrs.device = s2_device;
        end;

        memattrs.xs = s1_memattrs.xs;

    elsif s2_attr[1:0] == '11' then     // S2 attr = S1 attr
        memattrs = s1_memattrs;

    elsif s2_attr[1:0] == '10' then     // Force writeback
        memattrs.memtype = MemType\_Normal;
        memattrs.inner.attrs = MemAttr\_WB;
        memattrs.outer.attrs = MemAttr\_WB;

        if (s1_memattrs.memtype == MemType\_Normal &&
            s1_memattrs.inner.attrs != MemAttr\_NC) then
            memattrs.inner.hints = s1_memattrs.inner.hints;
            memattrs.inner.transient = s1_memattrs.inner.transient;
        else
            memattrs.inner.hints = MemHint\_RWA;
            memattrs.inner.transient = FALSE;
        end;

        if (s1_memattrs.memtype == MemType\_Normal &&
            s1_memattrs.outer.attrs != MemAttr\_NC) then
            memattrs.outer.hints = s1_memattrs.outer.hints;
            memattrs.outer.transient = s1_memattrs.outer.transient;
        else
            memattrs.outer.hints = MemHint\_RWA;
            memattrs.outer.transient = FALSE;
        end;

        memattrs.xs = '0';

    else                               // Non-cacheable unless S1 is device
        if s1_memattrs.memtype == MemType\_Device then
            memattrs = s1_memattrs;
        else
            var cacheability_attr : MemAttrHints;
            cacheability_attr.attrs = MemAttr\_NC;

            memattrs.memtype = MemType\_Normal;
            memattrs.inner = cacheability_attr;
            memattrs.outer = cacheability_attr;

            memattrs.xs = s1_memattrs.xs;
        end;
    end;

    let s2_shareability : Shareability = DecodeShareability(s2_sh);
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability, s2_shareability);
    memattrs.tags = S2MemTagType(memattrs, s1_memattrs.tags);
    memattrs.notagaccess = (s2_attr[3:1] == '111' && memattrs.tags == MemTag\_AllocationTagged);

    if s2_fnxs == '1' then
        memattrs.xs = '0';
    end;

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
end;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64_GetS1TLBContext

```
// AArch64_GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLB entries

func AArch64_GetS1TLBContext(regime : Regime, ss : SecurityState,
                             va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    case regime of
        when Regime\_EL3 => tlbcontext = AArch64\_TLBContextEL3(ss, va, tg);
        when Regime\_EL2 => tlbcontext = AArch64\_TLBContextEL2(ss, va, tg);
        when Regime\_EL20 => tlbcontext = AArch64\_TLBContextEL20(ss, va, tg);
        when Regime\_EL10 => tlbcontext = AArch64\_TLBContextEL10(ss, va, tg);
        otherwise =>
            unreachable;

    end;

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    tlbcontext.use_vmid = UseVMID(regime);
    // The following may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
end;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64_GetS2TLBContext

```
// AArch64_GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB entries

func AArch64_GetS2TLBContext(ss : SecurityState, ipa : FullAddress, tg : TGx) => TLBContext
begin
    assert EL2Enabled();

    var tlbcontext : TLBContext;

    tlbcontext.ss = ss;
    tlbcontext.regime = Regime\_EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid = VMID();
    tlbcontext.tg = tg;
    tlbcontext.ia = ZeroExtend{64}(ipa.address);
    if IsFeatureImplemented(FEAT_TTCNP) then
        tlbcontext.cnp = if ipa.paspace == PAS\_Secure then VSTTBR_EL2().CnP else VTTBR_EL2().CnP;
    else
        tlbcontext.cnp = '0';
    end;

    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.use_vmid = TRUE;
    // This may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
end;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64_TLBContextEL10

```
// AArch64_TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime to match against TLB entries

func AArch64_TLBContextEL10(ss : SecurityState, va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss      = ss;
    tlbcontext.regime  = Regime\_EL10;
    tlbcontext.vmid    = VMID();

    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() && TCR2_EL1().A2 == '1' then
        let varange : VARange = AArch64\_GetVARange(va);
        tlbcontext.asid = if varange == VARange\_LOWER then TTBR0_EL1().ASID else TTBR1_EL1().ASID;
    else
        tlbcontext.asid = if TCR_EL1().A1 == '0' then TTBR0_EL1().ASID else TTBR1_EL1().ASID;
    end;

    if TCR_EL1().AS == '0' then
        tlbcontext.asid[15:8] = Zeros{8};
    end;

    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;

    if IsFeatureImplemented(FEAT_TTCNP) then
        if AArch64\_GetVARange(va) == VARange\_LOWER then
            tlbcontext.cnp = TTBR0_EL1().CnP;
        else
            tlbcontext.cnp = TTBR1_EL1().CnP;
        end;
    else
        tlbcontext.cnp = '0';
    end;

    return tlbcontext;
end;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64_TLBContextEL2

```
// AArch64_TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries

func AArch64_TLBContextEL2(ss : SecurityState, va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss      = ss;
    tlbcontext.regime  = Regime\_EL2;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;
    tlbcontext.cnp     = if IsFeatureImplemented(FEAT_TTCNP) then TTBR0_EL2().CnP else '0';

    return tlbcontext;
end;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64_TLBContextEL20

```
// AArch64_TLBContextEL20()
// =====
// Gather translation context for accesses under EL20 regime to match against TLB entries

func AArch64_TLBContextEL20(ss : SecurityState, va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime\_EL20;

    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled() && TCR2_EL2().A2 == '1' then
        let varange : VARange = AArch64\_GetVARange(va);
        tlbcontext.asid = if varange == VARange\_LOWER then TTBR0_EL2().ASID else TTBR1_EL2().ASID;
    else
        tlbcontext.asid = if TCR_EL2().A1 == '0' then TTBR0_EL2().ASID else TTBR1_EL2().ASID;
    end;

    if TCR_EL2().AS == '0' then
        tlbcontext.asid[15:8] = Zeros{8};
    end;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;

    if IsFeatureImplemented(FEAT_TTCNP) then
        if AArch64\_GetVARange(va) == VARange\_LOWER then
            tlbcontext.cnp = TTBR0_EL2().CnP;
        else
            tlbcontext.cnp = TTBR1_EL2().CnP;
        end;
    else
        tlbcontext.cnp = '0';
    end;

    return tlbcontext;
end;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64_TLBContextEL3

```
// AArch64_TLBContextEL3()
// =====
// Gather translation context for accesses under EL3 regime to match against TLB entries

func AArch64_TLBContextEL3(ss : SecurityState, va : bits(64), tg : TGx) => TLBContext
begin
    var tlbcontext : TLBContext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime\_EL3;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;
    tlbcontext.cnp      = if IsFeatureImplemented(FEAT_TTCNP) then TTBR0_EL3().CnP else '0';

    return tlbcontext;
end;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64_FullTranslate

```
// AArch64_FullTranslate()
// =====
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation

func AArch64_FullTranslate(va : bits(64), size : integer, accdesc : AccessDescriptor,
                           aligned : boolean) => AddressDescriptor
begin
    let regime : Regime = TranslationRegime(accdesc.el);
    var fault : FaultRecord = NoFault(accdesc, va);

    var ipa : AddressDescriptor;
    (fault, ipa) = AArch64\_S1Translate(fault, regime, va, size, aligned, accdesc);

    if fault.statuscode != Fault\_None then
        return CreateFaultyAddressDescriptor(fault);
    end;

    if accdesc.ss == SS\_Realm then
        assert EL2Enabled();
    end;

    if regime == Regime\_EL10 && EL2Enabled() then
        let slaarch64 : boolean = TRUE;
        var pa : AddressDescriptor;
        (fault, pa) = AArch64\_S2Translate(fault, ipa, slaarch64, aligned, accdesc);

        if fault.statuscode != Fault\_None then
            return CreateFaultyAddressDescriptor(fault);
        else
            return pa;
        end;
    else
        return ipa;
    end;
end;
```


Library pseudocode for aarch64/translation/vmsa_translation/AArch64_MemSwapTableDesc

```
// AArch64_MemSwapTableDesc()
// =====
// Perform HW update of table descriptor as an atomic operation

func AArch64_MemSwapTableDesc(N : integer{64, 128})(fault_in : FaultRecord, prev_desc : bits(N),
    new_desc : bits(N), ee : bit,
    descaccess : AccessDescriptor,
    descpaddr : AddressDescriptor
) => (FaultRecord, bits(N))

begin
    assert descaccess.acctype == AccessType TTW;
    var fault : FaultRecord = fault_in;
    var iswrite : boolean;

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(descpaddr, descaccess);
        if fault.gpcf.gpf != GPCF None then
            fault.statuscode = Fault GPCFOnWalk;
            fault.paddress = descpaddr.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, ARBITRARY : bits(N));
        end;
    end;

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    var mem_desc : bits(N);
    var memstatus : PhysMemRetStatus;
    (memstatus, mem_desc) = PhysMemRead{N}(descpaddr, descaccess);

    if ee == '1' then
        mem_desc = BigEndianReverse{N}(mem_desc);
    end;

    if IsFault(memstatus) then
        iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, ARBITRARY : bits(N));
        end;
    end;

    if mem_desc == prev_desc then
        let ordered_new_desc : bits(N) = (if ee == '1' then BigEndianReverse{N}(new_desc)
            else new_desc);
        memstatus = PhysMemWrite{N}(descpaddr, descaccess, ordered_new_desc);

        if IsFault(memstatus) then
            iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8,
                fault);

            if IsFault(fault.statuscode) then
                return (fault, ARBITRARY : bits(N));
            end;
        end;

        // Reflect what is now in memory (in little endian format)
        mem_desc = new_desc;
    end;

    return (fault, mem_desc);
end;
```



```

// AArch64_S1DisabledOutput()
// =====
// Map the VA to IPA/PA and assign default memory attributes

func AArch64_S1DisabledOutput(fault_in : FaultRecord, regime : Regime,
                             va_in : bits(64), accdesc : AccessDescriptor,
                             aligned : boolean) => (FaultRecord, AddressDescriptor)

begin
    var va : bits(64) = va_in;
    let walkparams : S1TTWParams = AArch64\_GetS1TTWParams(regime, accdesc.el, accdesc.ss, va);
    var fault : FaultRecord = fault_in;

    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    // Output Address
    var oa : FullAddress;
    oa.address = va[NUM\_PABITS-1:0];
    case accdesc.ss of
        when SS\_Secure => oa.paspace = PAS\_Secure;
        when SS\_NonSecure => oa.paspace = PAS\_NonSecure;
        when SS\_Root => oa.paspace = PAS\_Root;
        when SS\_Realm => oa.paspace = PAS\_Realm;
    end;

    var memattrs : MemoryAttributes;
    if regime == Regime\_EL10 && EL2Enabled() && walkparams.dc == '1' then
        var default_cacheability : MemAttrHints;
        default_cacheability.attrs = MemAttr\_WB;
        default_cacheability.hints = MemHint\_RWA;
        default_cacheability.transient = FALSE;

        memattrs.memtype = MemType\_Normal;
        memattrs.outer = default_cacheability;
        memattrs.inner = default_cacheability;
        memattrs.shareability = Shareability\_NSH;
        if walkparams.dct == '1' then
            memattrs.tags = MemTag\_AllocationTagged;
        elsif IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) && walkparams.mtx == '1' then
            memattrs.tags = MemTag\_CanonicallyTagged;
        else
            memattrs.tags = MemTag\_Untagged;
        end;
        memattrs.xs = '0';
    elsif accdesc.acctype == AccessType\_IFETCH then
        var i_cache_attr : MemAttrHints;
        if AArch64\_S1ICacheEnabled(regime) then
            i_cache_attr.attrs = MemAttr\_WT;
            i_cache_attr.hints = MemHint\_RA;
            i_cache_attr.transient = FALSE;
        else
            i_cache_attr.attrs = MemAttr\_NC;
        end;

        memattrs.memtype = MemType\_Normal;
        memattrs.outer = i_cache_attr;
        memattrs.inner = i_cache_attr;
        memattrs.shareability = Shareability\_OSH;
        memattrs.tags = MemTag\_Untagged;
        memattrs.xs = '1';
    elsif accdesc.acctype == AccessType\_SPE && EffectivePMBLIMITR\_EL1\_nVM() == '1' then
        memattrs = S1DecodeMemAttrs(PMBMAR_EL1().Attr, PMBMAR_EL1().SH, TRUE,
                                     walkparams, accdesc.acctype);
    elsif accdesc.acctype == AccessType\_TRBE && EffectiveTRBLIMITR\_EL1\_nVM() == '1' then
        memattrs = S1DecodeMemAttrs(TRBMAR_EL1().Attr, TRBMAR_EL1().SH, TRUE,
                                     walkparams, accdesc.acctype);
    else
        memattrs.memtype = MemType\_Device;
        memattrs.device = DeviceType\_nGnRnE;
        memattrs.shareability = Shareability\_OSH;
        if IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) && walkparams.mtx == '1' then
            memattrs.tags = MemTag\_CanonicallyTagged;
        else
            memattrs.tags = MemTag\_Untagged;
        end;
        memattrs.xs = '1';
    end;
    memattrs.notagaccess = FALSE;

    if walkparams.mtx == '1' && walkparams.tbi == '0' && accdesc.acctype != AccessType\_IFETCH then
        // For the purpose of the checks in this function, the MTE tag bits are ignored.
        va[59:56] = if HasUnprivileged(regime) then Replicate{4}(va[55]) else '0000';
    end;

    fault.level = 0;
    let addrtop : AddressSize = AArch64\_AddrTop(walkparams.tbid, accdesc.acctype,
                                              walkparams.tbi);
    let pamax : AddressSize = AArch64\_PAMax();

```

```

if !IsZero(va[addrtop:pamax]) then
    fault.statuscode = Fault AddressSize;
elseif AArch64\_S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, walkparams.ntlsmid,
                                                memattrs) then
    fault.statuscode = Fault Alignment;
elseif ((accdesc.exclusive || accdesc.atomicop) &&
        !(regime == Regime\_EL10 && EL2Enabled() && HCR_EL2().DC == '1') &&
        ConstrainUnpredictableBool(Unpredictable Atomic MMU IMPDEF FAULT)) then
    fault.statuscode = Fault Exclusive;
end;

if fault.statuscode != Fault None then
    return (fault, ARBITRARY : AddressDescriptor);
else
    var ipa : AddressDescriptor = CreateAddressDescriptor(va_in, oa, memattrs, accdesc);
    ipa.mecid = AArch64\_S1DisabledOutputMECID(walkparams, regime, ipa.paddress.paspace);
    return (fault, ipa);
end;
end;

```



```

// AArch64_S1Translate()
// =====
// Translate VA to IPA/PA depending on the regime

func AArch64_S1Translate(fault_in : FaultRecord, regime : Regime,
                        va : bits(64), size : integer, aligned : boolean,
                        accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
    recurselimit Unbounded_DescriptorUpdate

begin
    var fault : FaultRecord = fault_in;
    // Prepare fault fields in case a fault is detected
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    if !AArch64\_S1Enabled(regime, accdesc.acctype) then
        return AArch64\_S1DisabledOutput(fault, regime, va, accdesc, aligned);
    end;

    var walkparams : S1TTWParams = AArch64\_GetS1TTWParams(regime, accdesc.el, accdesc.ss, va);
    let slmintxsx : integer = AArch64\_S1MinTxSZ(regime, walkparams);
    let slmaxtxsz : integer = AArch64\_MaxTxSZ(walkparams.tgx);
    if AArch64\_S1TxSZFaults(regime, walkparams) then
        fault.statuscode = Fault\_Translation;
        fault.level = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    elsif UInt(walkparams.txsz) < slmintxsx then
        walkparams.txsz = slmintxsx[5:0];
    elsif UInt(walkparams.txsz) > slmaxtxsz then
        walkparams.txsz = slmaxtxsz[5:0];
    end;

    if AArch64\_VAIsOutOfRange(va, accdesc.acctype, regime, walkparams) then
        fault.statuscode = Fault\_Translation;
        fault.level = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if accdesc.el == EL0 && walkparams.e0pd == '1' then
        fault.statuscode = Fault\_Translation;
        fault.level = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if (IsFeatureImplemented(FEAT_SVE) && accdesc.el == EL0 && walkparams.nfd == '1' &&
        ((accdesc.nonfault && accdesc.contiguous) ||
         (accdesc.firstfault && !accdesc.first && !accdesc.contiguous))) then
        fault.statuscode = Fault\_Translation;
        fault.level = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var descipaddr : AddressDescriptor;
    var walkstate : TTWState;
    var descriptor : bits(128);
    if walkparams.dl28 == '1' then
        (fault, descipaddr, walkstate, descriptor) = AArch64\_S1Walk{128}(fault, walkparams, va,
                                                                    regime, accdesc);
    else
        (fault, descipaddr, walkstate, descriptor[63:0]) = AArch64\_S1Walk{64}(fault, walkparams, va,
                                                                    regime, accdesc);
        descriptor[127:64] = Zeros{64};
    end;
    if fault.statuscode != Fault\_None then
        return (fault, ARBITRARY : AddressDescriptor);
    end;
    if AArch64\_S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, walkparams.ntlsm,
                                                walkstate.memattrs) then
        fault.statuscode = Fault\_Alignment;
    end;

    let fault_perm : FaultRecord = AArch64\_S1CheckPermissions(fault, va, size, regime,
                                                            walkstate, walkparams, accdesc);

    var mem_desc : bits(128);
    var new_desc : bits(128) = descriptor;

    if AArch64\_SetAccessFlag(walkparams.ha, accdesc, fault, fault_perm) then
        // Set descriptor AF bit
        new_desc[10] = '1';
    end;

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.

    if AArch64\_SetDirtyState(walkparams.hd, (walkparams.pie OR descriptor[51]),
                            accdesc, fault, fault_perm) then
        // Clear descriptor AP[2]/nDirty bit permitting stage 1 writes

```

```

    new_desc[7] = '0';
end;

if fault.statuscode == Fault\_None && fault_perm.statuscode != Fault\_None then
    fault = fault_perm;
end;

// Either the access flag was clear or AP[2]/nDirty was set
if new_desc != descriptor then
    if (!IsWBShareable(descipaddr.memattrs) &&
        ConstrainUnpredictableBool(Unpredictable\_Unsupported\_Atomic\_HW\_Update)) then
        fault.statuscode = Fault\_UnsupportedAtomicHWUpdate;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var descipaddr : AddressDescriptor;
    let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);

    if regime == Regime\_EL10 && EL2Enabled() then
        var s2fault : FaultRecord;
        let slaarch64 : boolean = TRUE;
        let s2aligned : boolean = TRUE;
        (s2fault, descipaddr) = AArch64\_S2Translate(fault, descipaddr, slaarch64, s2aligned,
            descaccess);

        if s2fault.statuscode != Fault\_None then
            return (s2fault, ARBITRARY : AddressDescriptor);
        end;

    else
        descipaddr = descipaddr;
    end;

    if walkparams.d128 == '1' then
        (fault, mem_desc) = AArch64\_MemSwapTableDesc{128}(fault, descriptor,
            new_desc, walkparams.ee,
            descaccess, descipaddr);

    else
        (fault, mem_desc[63:0]) = AArch64\_MemSwapTableDesc{64}(fault, descriptor[63:0],
            new_desc[63:0], walkparams.ee,
            descaccess, descipaddr);

        mem_desc[127:64] = Zeros{64};
    end;

    if fault.statuscode != Fault\_None then
        if (accdesc.acctype == AccessType\_AT &&
            !(ImpDefBool("AT reports the HW update fault"))) then
            // Mask the fault
            fault.statuscode = Fault\_None;
        else
            return (fault, ARBITRARY : AddressDescriptor);
        end;
    elsif new_desc != descriptor && mem_desc != new_desc then
        // HW update of Dirty state or AF was not successful due to the descriptor being updated
        // not matching the descriptor used for translation. Due to this, the walk is restarted.
        return AArch64\_S1Translate(fault_in, regime, va, size, aligned, accdesc);
    end;
end;

if fault.statuscode != Fault\_None then
    return (fault, ARBITRARY : AddressDescriptor);
end;

// Output Address
let oa : FullAddress = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);
var memattrs : MemoryAttributes;
if AArch64\_S1TreatAsNormalNC(walkstate, regime, accdesc) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;

    // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE on the Tagged attribute
    // when the memory region is Allocation Tagged.
    if (IsFeatureImplemented(FEAT_MTE2) &&
        walkstate.memattrs.tags == MemTag\_AllocationTagged &&
        ConstrainUnpredictableBool(Unpredictable\_S1CTAGGED)) then
        memattrs.tags = MemTag\_AllocationTagged;
    // SCTLR_ELx.C has no effect on whether the memory region is treated as Canonically Tagged.
    elsif (IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) &&
        walkstate.memattrs.tags == MemTag\_CanonicallyTagged) then
        memattrs.tags = MemTag\_CanonicallyTagged;
    end;
else
    memattrs = walkstate.memattrs;
end;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && EL2Enabled() && HCR_EL2().VM == '1' &&
    !(ImpDefBool("Apply effective shareability at stage 1"))) then
    memattrs.shareability = walkstate.memattrs.shareability;
end;

```

```

else
    memattrs.shareability = EffectiveShareability(memattrs);
end;

var ipa : AddressDescriptor = CreateAddressDescriptor(va, oa, memattrs, accdesc);
ipa.slassured = walkstate.slassured;
let varange : VARange = AArch64\_GetVARange(va);
ipa.mecid = AArch64\_S1OutputMECID{128}(walkparams, regime, varange, ipa.paddress.paspace,
    descriptor);

if (accdesc.atomicop && !IsWBShareable(memattrs) &&
    ConstrainUnpredictableBool(Unpredictable Atomic MMU IMPDEF FAULT)) then
    fault.statuscode = Fault Exclusive;
    return (fault, ipa);
end;

if accdesc.ls64 && memattrs.memtype == MemType Normal then
    if IsFeatureImplemented(FEAT_LS64WB) && !accdesc.withstatus then
        if (!IsWBShareable(memattrs) &&
            !(memattrs.inner.attrs == MemAttr NC &&
              memattrs.outer.attrs == MemAttr NC) &&
            (ImpDefBool(
                "LD64B or ST64B faults to cacheable non-iWBowB memory"))) then
            fault.statuscode = Fault Exclusive;
            return (fault, ipa);
        end;
    elsif !(memattrs.inner.attrs == MemAttr NC && memattrs.outer.attrs == MemAttr NC) then
        fault.statuscode = Fault Exclusive;
        return (fault, ipa);
    end;
end;

return (fault, ipa);
end;

```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64_S1TreatAsNormalNC

```

// AArch64_S1TreatAsNormalNC()
// =====
// Returns TRUE if stage 1 memory attributes should be treated as Normal Non-Cacheable

func AArch64_S1TreatAsNormalNC(walkstate : TTWState, regime : Regime,
    accdesc : AccessDescriptor) => boolean
begin
    return ((accdesc.acctype == AccessType IFETCH &&
        (walkstate.memattrs.memtype == MemType Device || !AArch64\_S1ICacheEnabled(regime))) ||
        (accdesc.acctype != AccessType IFETCH &&
            !AArch64\_S1DCacheEnabled(regime) && walkstate.memattrs.memtype == MemType Normal));
end;

```



```

// AArch64_S2Translate()
// =====
// Translate stage 1 IPA to PA and combine memory attributes

func AArch64_S2Translate(fault_in : FaultRecord, ipa : AddressDescriptor,
    slaarch64 : boolean, aligned : boolean,
    accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
    recurselimit Unbounded_DescriptorUpdate

begin
    var walkparams : S2TTWParams = AArch64\_GetS2TTWParams(accdesc.ss, ipa.paddress.paspace,
        slaarch64);

    var fault : FaultRecord = fault_in;
    var s2fslmro : boolean;
    // Prepare fault fields in case a fault is detected
    fault.statuscode = Fault\_None; // Ignore any faults from stage 1
    fault.dirtybit    = FALSE;
    fault.overlay     = FALSE;
    fault.tagaccess   = FALSE;
    fault.sitagnotdata = FALSE;
    fault.secondstage  = TRUE;
    fault.s2fslwalk   = accdesc.acctype == AccessType\_TTW;
    fault.ipaddress    = ipa.paddress;

    if walkparams.vm != '1' then
        // Stage 2 translation is disabled
        return (fault, ipa);
    end;

    let s2mintxsx : integer = AArch64\_S2MinTxSZ(walkparams, slaarch64);
    let s2maxtxsz : integer = AArch64\_MaxTxSZ(walkparams.tgx);
    if AArch64\_S2TxSZFaults(walkparams, slaarch64) then
        fault.statuscode = Fault\_Translation;
        fault.level      = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    elsif UInt(walkparams.txsz) < s2mintxsx then
        walkparams.txsz = s2mintxsx[5:0];
    elsif UInt(walkparams.txsz) > s2maxtxsz then
        walkparams.txsz = s2maxtxsz[5:0];
    end;

    if (walkparams.dl28 == '0' &&
        (AArch64\_S2InvalidSL(walkparams) || AArch64\_S2InconsistentSL(walkparams))) then
        fault.statuscode = Fault\_Translation;
        fault.level      = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if AArch64\_IPAIsOutOfRange(ipa.paddress.address, walkparams) then
        fault.statuscode = Fault\_Translation;
        fault.level      = 0;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    var descaddr : AddressDescriptor;
    var walkstate : TTWState;
    var descriptor : bits(128);
    if walkparams.dl28 == '1' then
        (fault, descaddr, walkstate, descriptor) = AArch64\_S2Walk{128}(fault, ipa, walkparams,
            accdesc);
    else
        (fault, descaddr, walkstate, descriptor[63:0]) = AArch64\_S2Walk{64}(fault, ipa, walkparams,
            accdesc);
        descriptor[127:64] = Zeros{64};
    end;
    if fault.statuscode != Fault\_None then
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if AArch64\_S2HasAlignmentFaultDueToMemType(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault\_Alignment;
    end;

    var fault_perm : FaultRecord;
    (fault_perm, s2fslmro) = AArch64\_S2CheckPermissions(fault, walkstate, walkparams, ipa, accdesc);

    var mem_desc : bits(128);
    var new_desc : bits(128) = descriptor;

    if AArch64\_SetAccessFlag(walkparams.ha, accdesc, fault, fault_perm) then
        // Set descriptor AF bit
        new_desc[10] = '1';
    end;

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.

    if AArch64\_SetDirtyState(walkparams.hd, (walkparams.s2pie OR descriptor[51]),

```

```

        accdesc, fault, fault_perm) then
    // Set descriptor S2AP[1]/Dirty bit permitting stage 2 writes
    new_desc[7] = '1';
end;

if fault.statuscode == Fault\_None && fault_perm.statuscode != Fault\_None then
    fault = fault_perm;
end;

// Either the access flag was clear or S2AP[1]/Dirty was clear
if new_desc != descriptor then
    if (!IsWBShareable(descpaddr.memattrs) &&
        ConstrainUnpredictableBool(Unpredictable\_Unsupported\_Atomic\_HW\_Update)) then
        fault.statuscode = Fault\_UnsupportedAtomicHWUpdate;
        return (fault, ARBITRARY : AddressDescriptor);
    end;

    if walkparams.hdbss == '1' && descriptor[7] == '0' && new_desc[7] == '1' then
        fault = AppendToHDBSS(fault, ipa.paddress, accdesc, walkparams, walkstate.level);
    end;

    // If an error, other than a synchronous External abort, occurred on the HDBSS update,
    // stage 2 hardware update of dirty state is not permitted.
    if (!(walkparams.hdbss == '1' && descriptor[7] != new_desc[7]) ||
        (HDBSSPROD_EL2().FSC != '101000' &&
         (!fault.hdbssf || IsExternalAbort(fault.statuscode)))) then
        let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);
        if walkparams.dl28 == '1' then
            (fault, mem_desc) = AArch64\_MemSwapTableDesc{128}(fault, descriptor, new_desc,
                                                             walkparams.ee, descaccess,
                                                             descpaddr);
        else
            (fault, mem_desc[63:0]) = AArch64\_MemSwapTableDesc{64}(fault, descriptor[63:0],
                                                                    new_desc[63:0], walkparams.ee,
                                                                    descaccess, descpaddr);
            mem_desc[127:64] = Zeros{64};
        end;
    end;
end;

if fault.statuscode != Fault\_None then
    if (accdesc.acctype == AccessType\_AT &&
        !(ImpDefBool("AT reports the HW update fault"))) then
        // Mask the fault
        fault.statuscode = Fault\_None;
    else
        return (fault, ARBITRARY : AddressDescriptor);
    end;
elseif new_desc != descriptor && mem_desc != new_desc then
    // HW update of Dirty state or AF was not successful due to the descriptor being updated
    // not matching the descriptor used for translation. Due to this, the walk is restarted.
    return AArch64\_S2Translate(fault_in, ipa, slaarch64, aligned, accdesc);
end;
end;

if fault.statuscode != Fault\_None then
    return (fault, ARBITRARY : AddressDescriptor);
end;

let ipa_64 : bits(64) = ZeroExtend{}(ipa.paddress.address);
// Output Address
let oa : FullAddress = StageOA(ipa_64, walkparams.dl28, walkparams.tgx, walkstate);
var s2_memattrs : MemoryAttributes;

if AArch64\_S2TreatAsNormalNC(walkstate, walkparams, accdesc) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_memattrs = NormalNCMemAttr();
    s2_memattrs.xs = walkstate.memattrs.xs;
    if walkstate.memattrs.tags == MemTag\_CanonicallyTagged then
        s2_memattrs.tags = MemTag\_CanonicallyTagged;
    end;
else
    s2_memattrs = walkstate.memattrs;
end;

let s2aarch64 : boolean = TRUE;
var memattrs : MemoryAttributes;
if walkparams.fwb == '0' then
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
else
    memattrs = s2_memattrs;
end;

var pa : AddressDescriptor = CreateAddressDescriptor(ipa.vaddress, oa, memattrs, accdesc);
pa.s2fslmro = s2fslmro;
pa.mecid = AArch64\_S2OutputMECID{128}(walkparams, pa.paddress.paspace, descriptor);

if (accdesc.acctype == AccessType\_TTW && accdesc.atomicop && !IsWBShareable(pa.memattrs) &&
    ConstrainUnpredictableBool(Unpredictable\_Unsupported\_Atomic\_HW\_Update)) then
    fault.statuscode = Fault\_UnsupportedAtomicHWUpdate;
end;

```

```

    return (fault, pa);
end;

if (accdesc.acctype != AccessType\_TTW && accdesc.atomicop && !IsWBShareable(pa.memattrs) &&
    ConstrainUnpredictableBool(Unpredictable Atomic MMU IMPDEF FAULT)) then
    fault.statuscode = Fault\_Exclusive;
    return (fault, pa);
end;

if accdesc.ls64 && s2_memattrs.memtype == MemType\_Normal then
    if IsFeatureImplemented(FEAT LS64WB) && !accdesc.withstatus then
        if (!IsWBShareable(s2_memattrs) &&
            !(s2_memattrs.inner.attrs == MemAttr\_NC &&
              s2_memattrs.outer.attrs == MemAttr\_NC) &&
            (ImpDefBool(
                "LD64B or ST64B faults to cacheable non-iWBowB memory"))) then
            fault.statuscode = Fault\_Exclusive;
            return (fault, pa);
        end;
    elseif !(s2_memattrs.inner.attrs == MemAttr\_NC && s2_memattrs.outer.attrs == MemAttr\_NC) then
        fault.statuscode = Fault\_Exclusive;
        return (fault, pa);
    end;
end;

return (fault, pa);
end;

```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64_S2TreatAsNormalNC

```

// AArch64_S2TreatAsNormalNC()
// =====
// Returns TRUE if stage 2 memory attributes should be treated as Normal Non-cacheable

func AArch64_S2TreatAsNormalNC(walkstate : TTWState, walkparams : S2TTWParams,
                                accdesc : AccessDescriptor) => boolean
begin
    return ((accdesc.acctype == AccessType\_TTW &&
        walkstate.memattrs.memtype == MemType\_Device && walkparams.ptw == '0') ||
        (accdesc.acctype == AccessType\_IFETCH &&
        (walkstate.memattrs.memtype == MemType\_Device || HCR_EL2().ID == '1')) ||
        (accdesc.acctype != AccessType\_IFETCH &&
        walkstate.memattrs.memtype == MemType\_Normal && !S2DCCacheEnabled()));
end;

```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64_SetAccessFlag

```

// AArch64_SetAccessFlag()
// =====
// Determine whether the access flag could be set by HW given the fault status

func AArch64_SetAccessFlag(ha : bit, accdesc : AccessDescriptor, fault : FaultRecord,
                            fault_perm : FaultRecord) => boolean
begin
    if ha == '0' || !AArch64\_SettingAccessFlagPermitted(fault, fault_perm) then
        return FALSE;
    elseif accdesc.acctype == AccessType\_AT then
        return ImpDefBool("AT updates AF");
    elseif accdesc.acctype IN {AccessType\_DC, AccessType\_IC} then
        return ImpDefBool("Generate access flag fault on IC/DC operations");
    else
        // Set descriptor AF bit
        return TRUE;
    end;
end;

```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64_SetDirtyState

```
// AArch64_SetDirtyState()
// =====
// Determine whether dirty state is required to be updated by HW given the fault status

func AArch64_SetDirtyState(hd : bits(1), dbm : bits(1),
                           accdesc : AccessDescriptor,
                           fault : FaultRecord,
                           fault_perm : FaultRecord) => boolean
begin
    if hd == '0' then
        return FALSE;
    elsif !AArch64\_SettingDirtyStatePermitted(fault, fault_perm) then
        return FALSE;
    elsif accdesc.acctype IN {AccessType\_AT, AccessType\_IC, AccessType\_DC} then
        return FALSE;
    elsif !accdesc.write then
        return FALSE;
    else
        return dbm == '1';
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64_SettingAccessFlagPermitted

```
// AArch64_SettingAccessFlagPermitted()
// =====
// Determine whether the access flag could be set by HW given the fault status

func AArch64_SettingAccessFlagPermitted(fault : FaultRecord, fault_perm : FaultRecord) => boolean
begin
    if fault.statuscode == Fault\_None && fault_perm.statuscode == Fault\_None then
        return TRUE;
    elsif fault.statuscode == Fault\_Alignment || fault_perm.statuscode == Fault\_Permission then
        return ConstrainUnpredictableBool(Unpredictable\_AFUPDATE);
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64_SettingDirtyStatePermitted

```
// AArch64_SettingDirtyStatePermitted()
// =====
// Determine whether the dirty state could be set by HW given the fault status

func AArch64_SettingDirtyStatePermitted(fault : FaultRecord, fault_perm : FaultRecord) => boolean
begin
    if fault_perm.statuscode != Fault\_None then
        return FALSE;
    elsif fault.statuscode == Fault\_None then
        return TRUE;
    elsif fault.statuscode == Fault\_Alignment then
        return ConstrainUnpredictableBool(Unpredictable\_DBUPDATE);
    else
        return FALSE;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64_TranslateAddress

```
// AArch64_TranslateAddress()
// =====
// Main entry point for translating an address

func AArch64_TranslateAddress(va : bits(64), accdesc : AccessDescriptor,
                             aligned : boolean, size : integer) => AddressDescriptor
begin
    if (SPESampleInFlight &&
        !(accdesc.acctype IN {AccessType\_IFETCH, AccessType\_SPE, AccessType\_TRBE})) then
        SPESStartCounter(SPESCounterPosTranslationLatency);
    end;

    var result : AddressDescriptor = AArch64\_FullTranslate(va, size, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType\_IFETCH then
        // For an instruction fetch, CheckDebug will be called
        // after the instruction is read from memory
        result.fault = AArch64\_CheckDebug(va, accdesc, size);
    end;

    if (IsFeatureImplemented(FEAT_RME) && !IsFault(result) &&
        IsGranuleProtectionCheckedAccess(accdesc)) then
        result.fault.gpcf = GranuleProtectionCheck(result, accdesc);

        if result.fault.gpcf.gpcf != GPCF\_None then
            result.fault.statuscode = Fault\_GPCFOnOutput;
            result.fault.paddress = result.paddress;
            result.fault.vaddress = result.vaddress;
        end;
    end;

    if SPESampleInFlight &&
        !(accdesc.acctype IN {AccessType\_IFETCH, AccessType\_TRBE, AccessType\_SPE}) then
        SPESStopCounter(SPESCounterPosTranslationLatency);
    end;

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend{64}(va);

    return result;
end;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64_TranslateTagAddress

```
// AArch64_TranslateTagAddress()
// =====
// Translate an address for accessing an Allocation Tag.

func AArch64_TranslateTagAddress(va : bits(64),
                                 accdesc_in : AccessDescriptor,
                                 aligned : boolean,
                                 size : integer) => (MemTagType, AddressDescriptor)
begin
    var accdesc : AccessDescriptor = accdesc_in;
    let taddrdesc : AddressDescriptor = AArch64\_TranslateAddress(va, accdesc, aligned, size);

    return (taddrdesc.memattrs.tags, taddrdesc);
end;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64_BlockDescSupported

```
// AArch64_BlockDescSupported()
// =====
// Determine whether a block descriptor is valid for the given granule size
// and level

func AArch64_BlockDescSupported(d128 : bit, ds : bit, tdx : TGx, level : integer) => boolean
begin
    case tdx of
        when TGx\_4KB => return ((level == 0 && (ds == '1' || d128 == '1')) ||
                                level == 1 ||
                                level == 2);
        when TGx\_16KB => return ((level == 1 && (ds == '1' || d128 == '1')) ||
                                level == 2);
        when TGx\_64KB => return ((level == 1 && (d128 == '1' || AArch64\_PAMax() >= 52)) ||
                                level == 2);
    end;
    return FALSE;
end;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64_ContiguousBit

```
// AArch64_ContiguousBit()
// =====
// Get the value of the contiguous bit

func AArch64_ContiguousBit{N}(tgx : TGx, dl28 : bit, level : integer, descriptor : bits(N)) => bit
begin
    if ImpDefBool("Treat Contiguous bit as 0") then
        return '0';
    end;

    if dl28 == '1' then
        if (tgx == TGx 64KB && level == 1) || (tgx == TGx 4KB && level == 0) then
            return '0'; // RES0
        else
            return descriptor[111];
        end;
    end;

    // When using TGx 64KB then the Contiguous bit is RES0 for Block
    // descriptors at level 1. For VMSAv8-64, level 1 Block descriptors
    // can exist only if FEAT_LPA is implemented.

    if tgx == TGx 64KB && level == 1 then
        return '0'; // RES0
    end;

    // When the Effective value of TCR_ELx.DS is '1',
    // the Contiguous bit is RES0 for all the following:
    // * For TGx 4KB, Block descriptors at level 0
    // * For TGx 16KB, Block descriptors at level 1
    // For VMSAv8-64, the above Block descriptors at the specified
    // levels can exist only if the Effective value of TCR_ELx.DS is 1.

    if tgx == TGx 16KB && level == 1 then
        return '0'; // RES0
    end;

    if tgx == TGx 4KB && level == 0 then
        return '0'; // RES0
    end;

    return descriptor[52];
end;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64_DecodeDescriptorType

```
// AArch64_DecodeDescriptorType()
// =====
// Determine whether the descriptor is a page, block or table

func AArch64_DecodeDescriptorType{N}(descriptor : bits(N), dl28 : bit, ds : bit,
                                     tgx : TGx, level : integer) => DescriptorType
begin
    if descriptor[0] == '0' then
        return DescriptorType Invalid;
    elsif dl28 == '1' then
        let skl : bits(2) = descriptor[110:109];
        if tgx IN {TGx 16KB, TGx 64KB} && UInt(skl) == 3 then
            return DescriptorType Invalid;
        end;

        let effective_level : integer = level + UInt(skl);
        if effective_level > FINAL\_LEVEL then
            return DescriptorType Invalid;
        elsif effective_level == FINAL\_LEVEL then
            return DescriptorType Leaf;
        else
            return DescriptorType Table;
        end;
    else
        if descriptor[1] == '1' then
            if level == FINAL\_LEVEL then
                return DescriptorType Leaf;
            else
                return DescriptorType Table;
            end;
        elsif descriptor[1] == '0' then
            if AArch64 BlockDescSupported(dl28, ds, tgx, level) then
                return DescriptorType Leaf;
            else
                return DescriptorType Invalid;
            end;
        end;
        unreachable;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64_S1ApplyOutputPerms

```
// AArch64_S1ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 1 page/block descriptors

func AArch64_S1ApplyOutputPerms(N) (permissions_in : Permissions, descriptor : bits(N),
                                     regime : Regime, walkparams : S1TTWParams) => Permissions
begin
    var permissions : Permissions = permissions_in;

    var pi_index : bits(4);
    if walkparams.pie == '1' then
        if walkparams.dl28 == '1' then
            pi_index = descriptor[118:115];
        else
            pi_index = descriptor[54:53]::descriptor[51]::descriptor[6];
        end;
        permissions.ppi = walkparams.pir[UInt(pi_index)*4];
        permissions.upi = walkparams.pire0[UInt(pi_index)*4];
        permissions.ndirty = descriptor[7];
    else
        if regime == Regime\_EL10 && EL2Enabled() && walkparams.nvl == '1' then
            permissions.ap[2:1] = descriptor[7]::'0';
            permissions.pxn = descriptor[54];
        elsif HasUnprivileged(regime) then
            permissions.ap[2:1] = descriptor[7:6];
            permissions.uxn = descriptor[54];
            permissions.pxn = descriptor[53];
        else
            permissions.ap[2:1] = descriptor[7]::'1';
            permissions.xn = descriptor[54];
        end;
        permissions.dbm = descriptor[51];
    end;
    if IsFeatureImplemented(FEAT_S1POE) then
        if walkparams.dl28 == '1' then
            permissions.po_index[3:0] = descriptor[124:121];
        else
            permissions.po_index[3:0] = '0'::descriptor[62:60];
        end;
    end;

    return permissions;
end;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64_S1ApplyTablePerms

```
// AArch64_S1ApplyTablePerms()
// =====
// Apply hierarchical permissions encoded in stage 1 table descriptors

func AArch64_S1ApplyTablePerms(permissions_in : Permissions, descriptor : bits(64),
                                regime : Regime, walkparams : S1TTWParams) => Permissions
begin
    var permissions : Permissions = permissions_in;
    var ap_table : bits(2);
    var pxn_table : bit;
    var uxn_table : bit;
    var xn_table : bit;
    if regime == Regime\_EL10 && EL2Enabled() && walkparams.nvl == '1' then
        ap_table = descriptor[62]::'0';
        pxn_table = descriptor[60];
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

    elsif HasUnprivileged(regime) then
        ap_table = descriptor[62:61];
        uxn_table = descriptor[60];
        pxn_table = descriptor[59];
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        ap_table = descriptor[62]::'0';
        xn_table = descriptor[60];
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;
    end;

    return permissions;
end;
```


Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64_S2ApplyOutputPerms

```
// AArch64_S2ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 2 page/block descriptors

func AArch64_S2ApplyOutputPerms{N}(descriptor : bits(N), walkparams : S2TTWParams) => Permissions
begin
    var permissions : Permissions;
    var s2pi_index : bits(4);
    if walkparams.s2pie == '1' then
        if walkparams.d128 == '1' then
            s2pi_index = descriptor[118:115];
        else
            s2pi_index = descriptor[54:53,51,6];
        end;
        permissions.s2pi = walkparams.s2pir[UInt(s2pi_index)*:4];
        permissions.s2dirty = descriptor[7];
    else
        permissions.s2ap = descriptor[7:6];
        if walkparams.d128 == '1' then
            permissions.s2xn = descriptor[118];
        else
            permissions.s2xn = descriptor[54];
        end;

        if IsFeatureImplemented(FEAT_XNX) then
            if walkparams.d128 == '1' then
                permissions.s2xnx = descriptor[117];
            else
                permissions.s2xnx = descriptor[53];
            end;
        else
            permissions.s2xnx = '0';
        end;

        permissions.dbm = descriptor[51];
    end;
    if IsFeatureImplemented(FEAT_S2POE) then
        if walkparams.d128 == '1' then
            permissions.s2po_index = descriptor[124:121];
        else
            permissions.s2po_index = descriptor[62:59];
        end;
    end;
    return permissions;
end;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64_nTFFaults

```
// AArch64_nTFFaults()
// =====
// Identify whether the nT bit in a block or table descriptor is effectively set
// causing a translation fault

func AArch64_nTFFaults{N}(d128 : bit, descriptor : bits(N)) => boolean
begin
    if !IsFeatureImplemented(FEAT_BBML1) then
        return FALSE;
    end;
    let nT : bit = if d128 == '1' then descriptor[6] else descriptor[16];

    return nT == '1' && ImpDefBool("nT bit causes Translation Fault");
end;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64_S1InitialTTWState

```
// AArch64_S1InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 1

func AArch64_S1InitialTTWState(walkparams : S1TTWParams, va : bits(64), regime : Regime,
                               ss : SecurityState) => TTWState
begin
    var walkstate : TTWState;
    var tablebase : FullAddress;
    var permissions : Permissions;
    var ttbr : bits(128);

    ttbr = AArch64\_S1TTBR(regime, va);
    case ss of
        when SS\_Secure => tablebase.paspace = PAS\_Secure;
        when SS\_NonSecure => tablebase.paspace = PAS\_NonSecure;
        when SS\_Root => tablebase.paspace = PAS\_Root;
        when SS\_Realm => tablebase.paspace = PAS\_Realm;
    end;

    tablebase.address = AArch64\_S1TTBaseAddress{128}(walkparams, regime, ttbr);

    permissions.ap_table = '00';
    if HasUnprivileged(regime) then
        permissions.uxn_table = '0';
        permissions.pxn_table = '0';
    else
        permissions.xn_table = '0';
    end;

    walkstate.baseaddress = tablebase;
    walkstate.level = AArch64\_S1StartLevel(walkparams);
    walkstate.istable = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG = if HasUnprivileged(regime) then '1' else '0';
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    walkstate.permissions = permissions;
    if regime == Regime\_EL10 && EL2Enabled() && HCR_EL2().VM == '1' then
        if ((AArch64\_GetVARange(va) == VARange\_LOWER && VTCR_EL2().TL0 == '1') ||
            (AArch64\_GetVARange(va) == VARange\_UPPER && VTCR_EL2().TL1 == '1')) then
            walkstate.slassured = TRUE;
        else
            walkstate.slassured = FALSE;
        end;
    else
        walkstate.slassured = FALSE;
    end;
    walkstate.disch = walkparams.disch;

    return walkstate;
end;
```



```

// AArch64_S1NextWalkStateLeaf()
// =====
// Decode stage 1 page or block descriptor as output to this stage of translation

func AArch64_S1NextWalkStateLeaf(N)(currentstate : TTWState, s2fslmro : boolean, regime : Regime,
                                   accdesc : AccessDescriptor, walkparams : S1TTWParams,
                                   descriptor : bits(N)) => TTWState

begin
    var nextstate : TTWState;
    var baseaddress : FullAddress;
    baseaddress.address = AArch64\_S1LeafBase(N)(descriptor, walkparams, currentstate.level);

    if currentstate.baseaddress.paspace == PAS\_Secure then
        // Determine PA space of the block from NS bit
        let ns : bit = if walkparams.d128 == '1' then descriptor[127] else descriptor[5];
        baseaddress.paspace = if ns == '0' then PAS\_Secure else PAS\_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS\_Root then
        // Determine PA space of the block from NSE and NS bits
        let ns : bit = if walkparams.d128 == '1' then descriptor[127] else descriptor[5];
        let nse : bit = descriptor[11];
        let nse2 : bit = '0'; // NSE2 has the Effective value of 0 within a PE.
        baseaddress.paspace = DecodePASpace(nse2, nse, ns);

        // If Secure state is not implemented, but RME is,
        // force Secure space accesses to Non-secure space
        if baseaddress.paspace == PAS\_Secure && !HaveSecureState() then
            baseaddress.paspace = PAS\_NonSecure;
        end;

    elsif (currentstate.baseaddress.paspace == PAS\_Realm &&
          regime IN {Regime\_EL2, Regime\_EL20}) then
        // Realm EL2 and EL2&0 regimes have a stage 1 NS bit
        let ns : bit = if walkparams.d128 == '1' then descriptor[127] else descriptor[5];
        baseaddress.paspace = if ns == '0' then PAS\_Realm else PAS\_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS\_Realm then
        // Realm EL1&0 regime does not have a stage 1 NS bit
        baseaddress.paspace = PAS\_Realm;
    else
        baseaddress.paspace = PAS\_NonSecure;
    end;

    nextstate.istable = FALSE;
    nextstate.level = currentstate.level;
    nextstate.baseaddress = baseaddress;

    var attrindx : bits(4);
    if walkparams.aie == '1' then
        if walkparams.d128 == '1' then
            attrindx = descriptor[5:2];
        else
            attrindx = descriptor[59,4:2];
        end;
    else
        attrindx = '0'::descriptor[4:2];
    end;

    var sh : bits(2);
    if walkparams.d128 == '1' then
        sh = descriptor[9:8];
    elsif walkparams.ds == '1' then
        sh = walkparams.sh;
    else
        sh = descriptor[9:8];
    end;
    let attr : bits(8) = AArch64\_MAIRAttr(UInt(attrindx), walkparams.mair2, walkparams.mair);
    let slaarch64 : boolean = TRUE;

    nextstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64, walkparams, accdesc.acctype);
    nextstate.permissions = AArch64\_S1ApplyOutputPerms(N)(currentstate.permissions,
                                                         descriptor, regime, walkparams);

    var protectedbit : bit;
    if walkparams.d128 == '1' then
        protectedbit = descriptor[114];
    else
        protectedbit = if walkparams.pnch == '1' then descriptor[52] else '0';
    end;
    if (currentstate.slassured && s2fslmro && protectedbit == '1') then
        nextstate.slassured = TRUE;
    else
        nextstate.slassured = FALSE;
    end;

    if walkparams.pnch == '1' || currentstate.disch == '1' then
        nextstate.contiguous = '0';
    else
        nextstate.contiguous = AArch64\_ContiguousBit(N)(walkparams.tgx, walkparams.d128,
                                                         currentstate.level, descriptor);
    end;
    if !HasUnprivileged(regime) then

```

```

    nextstate.nG = '0';
elseif accdesc.ss == SS Secure && currentstate.baseaddress.paspace == PAS NonSecure then
    // In Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk
    nextstate.nG = '1';
elseif walkparams.fng == '1' then
    // Translations are treated as non-global regardless of the value of the nG bit.
    nextstate.nG = '1';
elseif (regime == Regime EL10 && EL2Enabled() && HCR_EL2().VM == '1' &&
        (walkparams.d128 == '1' || walkparams.pnch == '1') &&
        !nextstate.slassured && walkparams.fngna == '1') then
    // Translations are treated as non-global regardless of the value of the nG bit.
    nextstate.nG = '1';
else
    nextstate.nG = descriptor[11];
end;

if walkparams.d128 == '1' then
    nextstate.guardedpage = descriptor[113];
else
    nextstate.guardedpage = descriptor[50];
end;

return nextstate;
end;

```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64_S1NextWalkStateTable

```

// AArch64_S1NextWalkStateTable()
// =====
// Decode stage 1 table descriptor to transition to the next level

func AArch64_S1NextWalkStateTable(N)(currentstate : TTWState, s2fslmro : boolean, regime : Regime,
    walkparams : S1TTWParams, descriptor : bits(N)) => TTWState
begin
    var nextstate : TTWState;
    var tablebase : FullAddress;
    let skl : bits(2) = if walkparams.d128 == '1' then descriptor[110:109] else '00';
    tablebase.address = AArch64\_NextTableBase(N)(descriptor, walkparams.d128, skl, walkparams.ds,
        walkparams.tgx);

    if currentstate.baseaddress.paspace == PAS Secure then
        // Determine PA space of the next table from NSTable bit
        var nstable : bit;
        nstable = if walkparams.d128 == '1' then descriptor[127] else descriptor[63];
        tablebase.paspace = if nstable == '0' then PAS Secure else PAS NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;
    end;

    nextstate.istable = TRUE;
    nextstate.nG = currentstate.nG;
    if walkparams.d128 == '1' then
        nextstate.level = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level = currentstate.level + 1;
    end;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs = currentstate.memattrs;
    if walkparams.hpd == '0' && walkparams.pie == '0' then
        nextstate.permissions = AArch64\_S1ApplyTablePerms(currentstate.permissions,
            descriptor[63:0], regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;
    end;
    var protectedbit : bit;
    if walkparams.d128 == '1' then
        protectedbit = descriptor[114];
    else
        protectedbit = if walkparams.pnch == '1' then descriptor[52] else '0';
    end;
    if (currentstate.slassured && s2fslmro && protectedbit == '1') then
        nextstate.slassured = TRUE;
    else
        nextstate.slassured = FALSE;
    end;
    nextstate.disch = if walkparams.d128 == '1' then descriptor[112] else '0';

    return nextstate;
end;

```



```

// AArch64_S1Walk()
// =====
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

func AArch64_S1Walk(N : integer{64, 128})(fault_in : FaultRecord,
                                           walkparams : S1TTWParams,
                                           va : bits(64), regime : Regime,
                                           accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor, TTWState, bits(N))

begin
    var fault : FaultRecord = fault_in;
    var aligned : boolean;

    if HasUnprivileged(regime) && AArch64\_S1EPD(regime, va) == '1' then
        fault.statuscode = Fault\_Translation;
        fault.level = 0;
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
    end;

    var walkstate : TTWState = AArch64\_S1InitialTTWState(walkparams, va, regime, accdesc.ss);
    let startlevel : integer = walkstate.level;

    if startlevel > 3 then
        fault.statuscode = Fault\_Translation;
        fault.level = 0;
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
    end;

    var descriptor : bits(N);
    var walkaddress : AddressDescriptor;
    var skl : bits(2) = '00';
    walkaddress.vaddress = va;
    walkaddress.mecid = AArch64\_S1TTWalkMECID(walkparams.emec, regime, accdesc.ss);

    if !AArch64\_S1DCacheEnabled(regime) then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;
    end;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime\_EL10 && EL2Enabled() && HCR_EL2().VM == '1' &&
        !(ImpDefBool("Apply effective shareability at stage 1"))) then
        walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
    else
        walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);
    end;

    var s2fslmro : boolean = FALSE;

    var desctype : DescriptorType;
    var descaddress : FullAddress = AArch64\_S1SLTTEEntryAddress(walkstate.level, walkparams, va,
                                                             walkstate.baseaddress);

    // Detect Address Size Fault by Descriptor Address
    if AArch64\_S1OAOOutOfRange(descaddress.address, walkparams) then
        fault.statuscode = Fault\_AddressSize;
        fault.level = 0;
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
    end;

    repeat
        fault.level = walkstate.level;
        walkaddress.paddress = descaddress;
        walkaddress.slassured = walkstate.slassured;

        let toplevel : boolean = walkstate.level == startlevel;
        let varange : VARange = AArch64\_GetVARange(va);
        let walkaccess : AccessDescriptor = CreateAccDescS1TTW(toplevel, varange, accdesc);
        var s2fault : FaultRecord;
        var s2walkaddress : AddressDescriptor;
        if regime == Regime\_EL10 && EL2Enabled() then
            let slaarch64 : boolean = TRUE;
            aligned = TRUE;
            (s2fault, s2walkaddress) = AArch64\_S2Translate(fault, walkaddress, slaarch64, aligned,
                                                         walkaccess);

            if s2fault.statuscode != Fault\_None then
                return (s2fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                        ARBITRARY : bits(N));
            end;

            s2fslmro = s2walkaddress.s2fslmro;
            (fault, descriptor) = FetchDescriptor{N}(walkparams.ee, s2walkaddress, walkaccess,
                                                    fault);
        else

```

```

(fault, descriptor) = FetchDescriptor{N}(walkparams.ee, walkaddress, walkaccess,
                                         fault);

end;

if fault.statuscode != Fault None then
    return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
           ARBITRARY : bits(N));
end;

var new_descriptor : bits(N);
repeat
    new_descriptor = descriptor;
    desctype = AArch64 DecodeDescriptorType{N}(descriptor, walkparams.dl28, walkparams.ds,
                                                walkparams.tgx, walkstate.level);

    case desctype of
        when DescriptorType Table =>
            walkstate = AArch64 S1NextWalkStateTable{N}(walkstate, s2fslmro,
                                                         regime, walkparams, descriptor);
            skl = if walkparams.dl28 == '1' then descriptor[110:109] else '00';
            descaddress = AArch64 S1TTEntryAddress{N}(walkstate.level, walkparams, skl, va,
                                                    walkstate.baseaddress, descriptor);

            // Detect Address Size Fault by Descriptor Address
            if AArch64 S1OAOutOfRange(descaddress.address, walkparams) then
                fault.statuscode = Fault AddressSize;
                return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                       ARBITRARY : bits(N));
            end;

            if walkparams.haft == '1' then
                new_descriptor[10] = '1';
            end;

            if (walkparams.dl28 == '1' && skl != '00' &&
                AArch64 nTFaults{N}(walkparams.dl28, descriptor)) then
                fault.statuscode = Fault Translation;
                return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                       ARBITRARY : bits(N));
            end;

        when DescriptorType Leaf =>
            walkstate = AArch64 S1NextWalkStateLeaf{N}(walkstate, s2fslmro, regime, accdesc,
                                                         walkparams, descriptor);

        when DescriptorType Invalid =>
            fault.statuscode = Fault Translation;
            return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                   ARBITRARY : bits(N));

        otherwise =>
            unreachable;
    end;

    if new_descriptor != descriptor then

        if (!IsWBShareable(walkaddress.memattrs) &&
            ConstrainUnpredictableBool(Unpredictable Unsupported Atomic HW Update)) then
            fault.statuscode = Fault UnsupportedAtomicHWUpdate;
            return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                   ARBITRARY : bits(N));
        end;

        var descpaddr : AddressDescriptor;
        let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);
        if regime == Regime EL10 && EL2Enabled() then
            let slaarch64 : boolean = TRUE;
            aligned = TRUE;
            (s2fault, descpaddr) = AArch64 S2Translate(fault, walkaddress,
                                                         slaarch64, aligned,
                                                         descaccess);

            if s2fault.statuscode != Fault None then
                return (s2fault, ARBITRARY : AddressDescriptor,
                       ARBITRARY : TTWState, ARBITRARY : bits(N));
            end;
        else
            descpaddr = walkaddress;
        end;

        (fault, descriptor) = AArch64 MemSwapTableDesc{N}(fault, descriptor, new_descriptor,
                                                         walkparams.ee, descaccess,
                                                         descpaddr);

        if fault.statuscode != Fault None then
            return (fault, ARBITRARY : AddressDescriptor,
                   ARBITRARY : TTWState, ARBITRARY : bits(N));
        end;
    end;

until new_descriptor == descriptor looplimit Unbounded_DescriptorUpdate;
until desctype == DescriptorType Leaf looplimit MAX_WALK_LEVELS;

let oa : FullAddress = StageOA(va, walkparams.dl28, walkparams.tgx, walkstate);
if (walkstate.contiguous == '1' &&
    AArch64 ContiguousBitFaults(walkparams.dl28, walkparams.txsz, walkparams.tgx,

```



```

        walkstate.level)) then
    fault.statuscode = Fault Translation;
elseif walkstate.level < FINAL\_LEVEL && AArch64\_nTFaults{N}(walkparams.dl28, descriptor) then
    fault.statuscode = Fault Translation;
elseif AArch64\_S1AMECFault{N}(walkparams, walkstate.baseaddress.paspace, regime, descriptor) then
    fault.statuscode = Fault Translation;
// Detect Address Size Fault by final output
elseif AArch64\_S1OAOutOfRange(oa.address, walkparams) then
    fault.statuscode = Fault AddressSize;
// Check descriptor AF bit
elseif (descriptor[10] == '0' && walkparams.ha == '0' &&
        (!accdesc.acctype IN {AccessType\_DC, AccessType\_IC} ||
        ImpDefBool("Generate access flag fault on IC/DC operations"))) then
    fault.statuscode = Fault AccessFlag;
end;

if fault.statuscode != Fault None then
    return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
end;

return (fault, walkaddress, walkstate, descriptor);
end;

```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64_S2InitialTTWState

```

// AArch64_S2InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 2

func AArch64_S2InitialTTWState(ss : SecurityState, walkparams : S2TTWParams) => TTWState
begin
    var walkstate : TTWState;
    var tablebase : FullAddress;
    var ttbr : bits(128);

    ttbr = ZeroExtend{128}(VTBR_EL2());
    case ss of
        when SS\_NonSecure => tablebase.paspace = PAS\_NonSecure;
        when SS\_Realm    => tablebase.paspace = PAS\_Realm;
    end;
    tablebase.address = AArch64\_S2TTBaseAddress{128}(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64\_S2StartLevel(walkparams);
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
end;

```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64_S2NextWalkStateLeaf

```
// AArch64_S2NextWalkStateLeaf()
// =====
// Decode stage 2 page or block descriptor as output to this stage of translation

func AArch64_S2NextWalkStateLeaf{N}(currentstate : TTWState, ss : SecurityState,
                                     walkparams : S2TTWParams, ipa : AddressDescriptor,
                                     descriptor : bits(N)) => TTWState

begin
  var nextstate : TTWState;
  var baseaddress : FullAddress;

  if ss == SS_Secure then
    baseaddress.paspace = AArch64_SS2OutputPASpace(walkparams, ipa.paddress.paspace);
  elsif ss == SS_Realm then
    var ns : bit;
    ns = if walkparams.d128 == '1' then descriptor[127] else descriptor[55];
    baseaddress.paspace = if ns == '1' then PAS_NonSecure else PAS_Realm;
  else
    baseaddress.paspace = PAS_NonSecure;
  end;
  baseaddress.address = AArch64_S2LeafBase{N}(descriptor, walkparams, currentstate.level);

  nextstate.istable = FALSE;
  nextstate.level = currentstate.level;
  nextstate.baseaddress = baseaddress;
  nextstate.permissions = AArch64_S2ApplyOutputPerms{N}(descriptor, walkparams);

  let s2_attr : bits(4) = descriptor[5:2];
  let s2_sh : bits(2) = if walkparams.ds == '1' then walkparams.sh else descriptor[9:8];
  let s2_fnxs : bit = descriptor[11];
  if walkparams.fwb == '1' then
    nextstate.memattrs = AArch64_S2ApplyFWBMemAttrs{N}(ipa.memattrs, walkparams, descriptor);
    if s2_attr[3:1] == '111' then
      nextstate.permissions.s2tag_na = '1';
    else
      nextstate.permissions.s2tag_na = '0';
    end;
  else
    let s2aarch64 : boolean = TRUE;
    nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh, s2aarch64);
    // FnXS is used later to mask the XS value from stage 1
    nextstate.memattrs.xs = NOT s2_fnxs;
    if s2_attr == '0100' then
      nextstate.permissions.s2tag_na = '1';
    else
      nextstate.permissions.s2tag_na = '0';
    end;
  end;
  end;
  nextstate.contiguous = AArch64_ContiguousBit{N}(walkparams.tgx, walkparams.d128,
                                                    currentstate.level, descriptor);

  if walkparams.d128 == '1' then
    nextstate.s2assuredonly = descriptor[114];
  else
    nextstate.s2assuredonly = if walkparams.assuredonly == '1' then descriptor[58] else '0';
  end;

  return nextstate;
end;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64_S2NextWalkStateTable

```
// AArch64_S2NextWalkStateTable()
// =====
// Decode stage 2 table descriptor to transition to the next level

func AArch64_S2NextWalkStateTable(N)(currentstate : TTWState, walkparams : S2TTWParams,
                                     descriptor : bits(N)) => TTWState
begin
    var nextstate : TTWState;
    var tablebase : FullAddress;
    let skl : bits(2) = if walkparams.d128 == '1' then descriptor[110:109] else '00';
    tablebase.address = AArch64\_NextTableBase(N)(descriptor, walkparams.d128, skl, walkparams.ds,
                                             walkparams.tgx);

    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable      = TRUE;
    if walkparams.d128 == '1' then
        nextstate.level    = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level    = currentstate.level + 1;
    end;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;

    return nextstate;
end;
```



```

// AArch64_S2Walk()
// =====
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

func AArch64_S2Walk(N : integer{64, 128})(fault_in : FaultRecord,
                                           ipa : AddressDescriptor,
                                           walkparams : S2TTWParams,
                                           accdesc : AccessDescriptor) => (FaultRecord,
                                                                    AddressDescriptor,
                                                                    TTWState,
                                                                    bits(N))

begin

    var fault : FaultRecord = fault_in;
    let ipa_64 : bits(64) = ZeroExtend{}(ipa.paddress.address);

    var walkstate : TTWState;
    if accdesc.ss == SS\_Secure then
        walkstate = AArch64\_SS2InitialTTWState(walkparams, ipa.paddress.paspace);
    else
        walkstate = AArch64\_S2InitialTTWState(accdesc.ss, walkparams);
    end;
    let startlevel : integer = walkstate.level;

    if startlevel > 3 then
        fault.statuscode = Fault\_Translation;
        fault.level = 0;
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
    end;

    var descriptor : bits(N);
    let walkaccess : AccessDescriptor = CreateAccDescS2TTW(accdesc);
    var walkaddress : AddressDescriptor;
    var skl : bits(2) = '00';

    walkaddress.vaddress = ipa.vaddress;
    walkaddress.mecid = AArch64\_S2TTWalkMECID(walkparams.emec, accdesc.ss);

    if !S2DCacheEnabled() then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;
    end;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    var desctype : DescriptorType;

    // Initial lookup might index into concatenated tables
    var descaddress : FullAddress = AArch64\_S2SLTTEEntryAddress(walkparams, ipa.paddress.address,
                                                            walkstate.baseaddress);

    // Detect Address Size Fault by Descriptor Address
    if AArch64\_S2OAOutOfRange(descaddress.address, walkparams) then
        fault.statuscode = Fault\_AddressSize;
        fault.level = 0;
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState, ARBITRARY : bits(N));
    end;

    repeat
        fault.level = walkstate.level;
        walkaddress.paddress = descaddress;
        (fault, descriptor) = FetchDescriptor{N}(walkparams.ee, walkaddress, walkaccess, fault);

        if fault.statuscode != Fault\_None then
            return (fault, ARBITRARY : AddressDescriptor,
                    ARBITRARY : TTWState, ARBITRARY : bits(N));
        end;

        var new_descriptor : bits(N);
        repeat
            new_descriptor = descriptor;
            desctype = AArch64\_DecodeDescriptorType{N}(descriptor, walkparams.dl28, walkparams.ds,
                                                         walkparams.tgx, walkstate.level);

            case desctype of
                when DescriptorType\_Table =>
                    walkstate = AArch64\_S2NextWalkStateTable{N}(walkstate, walkparams, descriptor);
                    skl = if walkparams.dl28 == '1' then descriptor[110:109] else '00';
                    descaddress = AArch64\_S2TTEEntryAddress(walkstate.level, walkparams, skl,
                                                            ipa.paddress.address,
                                                            walkstate.baseaddress);

                    // Detect Address Size Fault by table descriptor
                    if AArch64\_S2OAOutOfRange(descaddress.address, walkparams) then
                        fault.statuscode = Fault\_AddressSize;
                        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                                ARBITRARY : bits(N));
                    end;
            end;
        repeat

```

```

end;

if walkparams.haft == '1' then
    new_descriptor[10] = '1';
end;

if (walkparams.d128 == '1' && skl != '00' &&
    AArch64_nTFaults{N}(walkparams.d128, descriptor)) then
    fault.statuscode = Fault_Translation;
    return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
            ARBITRARY : bits(N));
end;

when DescriptorType_Leaf =>
    walkstate = AArch64_S2NextWalkStateLeaf{N}(walkstate, accdesc.ss, walkparams,
                                                ipa, descriptor);

when DescriptorType_Invalid =>
    fault.statuscode = Fault_Translation;
    return (fault, ARBITRARY : AddressDescriptor,
            ARBITRARY : TTWState, ARBITRARY : bits(N));

otherwise =>
    unreachable;
end;

if new_descriptor != descriptor then

    if (!IsWBShareable(walkaddress.memattr) &&
        ConstrainUnpredictableBool(Unpredictable_Unsupported_Atomic_HW_Update)) then
        fault.statuscode = Fault_UnsupportedAtomicHWUpdate;
        return (fault, ARBITRARY : AddressDescriptor, ARBITRARY : TTWState,
                ARBITRARY : bits(N));
    end;

    let descaccess : AccessDescriptor = CreateAccDescTTEUpdate(accdesc);
    (fault, descriptor) = AArch64_MemSwapTableDesc{N}(fault, descriptor, new_descriptor,
                                                        walkparams.ee, descaccess,
                                                        walkaddress);

    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : AddressDescriptor,
                ARBITRARY : TTWState, ARBITRARY : bits(N));
    end;

end;

until new_descriptor == descriptor looplimit Unbounded_DescriptorUpdate;
until descctype == DescriptorType_Leaf looplimit MAX_WALK_LEVELS;

let oa : FullAddress = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
if (walkstate.contiguous == '1' &&
    AArch64_ContiguousBitFaults(walkparams.d128, walkparams.txs, walkparams.tgx,
                                walkstate.level)) then
    fault.statuscode = Fault_Translation;
elseif walkstate.level < FINAL_LEVEL && AArch64_nTFaults{N}(walkparams.d128, descriptor) then
    fault.statuscode = Fault_Translation;
// Detect Address Size Fault by final output
elseif AArch64_S2OAOutOfRange(oa.address, walkparams) then
    fault.statuscode = Fault_AddressSize;
// Check descriptor AF bit
elseif (descriptor[10] == '0' && walkparams.ha == '0' &&
        (!accdesc.acctype IN {AccessType_DC, AccessType_IC} ||
         ImpDefBool("Generate access flag fault on IC/DC operations"))) then
    fault.statuscode = Fault_AccessFlag;
end;

return (fault, walkaddress, walkstate, descriptor);
end;

```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64_SS2InitialTTWState

```
// AArch64_SS2InitialTTWState()
// =====
// Set properties of first access to translation tables in Secure stage 2

func AArch64_SS2InitialTTWState(walkparams : S2TTWParams, ipaspace : PASpace) => TTWState
begin
    var walkstate : TTWState;
    var tablebase : FullAddress;
    var ttbr : bits(128);

    if ipaspace == PAS\_Secure then
        ttbr = ZeroExtend{128}(VSTTBR_EL2());
    else
        ttbr = ZeroExtend{128}(VTTBR_EL2());
    end;

    if ipaspace == PAS\_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS\_Secure;
        else
            tablebase.paspace = PAS\_NonSecure;
        end;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = PAS\_Secure;
        else
            tablebase.paspace = PAS\_NonSecure;
        end;
    end;

    tablebase.address = AArch64\_S2TTBaseAddress{128}(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level        = AArch64\_S2StartLevel(walkparams);
    walkstate.istable      = TRUE;
    walkstate.memattrs     = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
end;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64_SS2OutputPASpace

```
// AArch64_SS2OutputPASpace()
// =====
// Assign PA Space to output of Secure stage 2 translation

func AArch64_SS2OutputPASpace(walkparams : S2TTWParams, ipaspace : PASpace) => PASpace
begin
    if ipaspace == PAS\_Secure then
        if walkparams.[sw,sa] == '00' then
            return PAS\_Secure;
        else
            return PAS\_NonSecure;
        end;
    else
        if walkparams.[sw,sa,nsw,nsa] == '0000' then
            return PAS\_Secure;
        else
            return PAS\_NonSecure;
        end;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_GetS1TTWParams

```
// AArch64_GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

func AArch64_GetS1TTWParams(regime : Regime, el : bits(2),
                             ss : SecurityState, va : bits(64)) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    let varange : VARange = AArch64_GetVARange(va);

    case regime of
        when Regime_EL3 => walkparams = AArch64_S1TTWParamsEL3();
        when Regime_EL2 => walkparams = AArch64_S1TTWParamsEL2(ss);
        when Regime_EL20 => walkparams = AArch64_S1TTWParamsEL20(el, ss, varange);
        when Regime_EL10 => walkparams = AArch64_S1TTWParamsEL10(el, varange);
    end;

    return walkparams;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_GetS2TTWParams

```
// AArch64_GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

func AArch64_GetS2TTWParams(ss : SecurityState, ipaspace : PASpace,
                             slaarch64 : boolean) => S2TTWParams
begin
    var walkparams : S2TTWParams;

    if ss == SS_NonSecure then
        walkparams = AArch64_NSS2TTWParams(slaarch64);
    elsif IsFeatureImplemented(FEAT_SEL2) && ss == SS_Secure then
        walkparams = AArch64_SS2TTWParams(ipaspace, slaarch64);
    elsif ss == SS_Realm then
        walkparams = AArch64_RLS2TTWParams(slaarch64);
    else
        unreachable;
    end;

    return walkparams;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_GetVARange

```
// AArch64_GetVARange()
// =====
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

func AArch64_GetVARange(va : bits(64)) => VARange
begin
    if va[55] == '0' then
        return VARange_LOWER;
    else
        return VARange_UPPER;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_HaveS1TG

```
// AArch64_HaveS1TG()
// =====
// Determine whether the given translation granule is supported for stage 1

func AArch64_HaveS1TG(tgx : TGx) => boolean
begin
    case tgx of
        when TGx_4KB => return IsFeatureImplemented(FEAT_TGran4K);
        when TGx_16KB => return IsFeatureImplemented(FEAT_TGran16K);
        when TGx_64KB => return IsFeatureImplemented(FEAT_TGran64K);
    end;
end;
```


Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_HaveS2TG

```
// AArch64_HaveS2TG()
// =====
// Determine whether the given translation granule is supported for stage 2

func AArch64_HaveS2TG(tgx : TGx) => boolean
begin
    assert HaveEL(EL2);

    if IsFeatureImplemented(FEAT_GTG) then
        case tgx of
            when TGx_4KB => return IsFeatureImplemented(FEAT_S2TGran4K);
            when TGx_16KB => return IsFeatureImplemented(FEAT_S2TGran16K);
            when TGx_64KB => return IsFeatureImplemented(FEAT_S2TGran64K);
        end;
    else
        return AArch64_HaveS1TG(tgx);
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_MaxTxSZ

```
// AArch64_MaxTxSZ()
// =====
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

func AArch64_MaxTxSZ(tgx : TGx) => integer
begin
    if IsFeatureImplemented(FEAT_TTST) then
        case tgx of
            when TGx_4KB => return 48;
            when TGx_16KB => return 48;
            when TGx_64KB => return 47;
        end;
    end;

    return 39;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_NSS2TTWParams

```
// AArch64_NSS2TTWParams()
// =====
// Gather walk parameters specific for Non-secure stage 2 translation

func AArch64_NSS2TTWParams(slaarch64 : boolean) => S2TTWParams
begin
    var walkparams : S2TTWParams;

    walkparams.vm = HCR_EL2().VM OR HCR_EL2().DC;
    walkparams.tgx = AArch64\_S2DecodeTG0(VTCR_EL2().TG0);
    walkparams.txsZ = VTCR_EL2().TOSZ;
    walkparams.ps = VTCR_EL2().PS;
    walkparams.irgn = VTCR_EL2().IRGN0;
    walkparams.orgn = VTCR_EL2().ORGN0;
    walkparams.sh = VTCR_EL2().SH0;
    walkparams.ee = SCTLR_EL2().EE;
    walkparams.d128 = if IsFeatureImplemented(FEAT_D128) then VTCR_EL2().D128 else '0';
    if walkparams.d128 == '1' then
        walkparams.skl = VTTBR_EL2().SKL;
    else
        walkparams.sl0 = VTCR_EL2().SL0;
    end;

    walkparams.ptw = if HCR_EL2().TGE == '0' then HCR_EL2().PTW else '0';
    walkparams.fwb = if IsFeatureImplemented(FEAT_S2FWB) then HCR_EL2().FWB else '0';
    walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then VTCR_EL2().HA else '0';
    if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
        walkparams.hd = VTCR_EL2().HD;
    else
        walkparams.hd = '0';
    end;
    if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = VTCR_EL2().DS;
    else
        walkparams.ds = '0';
    end;
    if walkparams.tgx == TGx\_4KB && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.sl2 = VTCR_EL2().SL2 AND VTCR_EL2().DS;
    else
        walkparams.sl2 = '0';
    end;
    walkparams.cmov = (if IsFeatureImplemented(FEAT_CMOW) && IsHCRXEL2Enabled() then HCRX_EL2().CMOW
        else '0');
    if walkparams.d128 == '1' then
        walkparams.s2pie = '1';
    else
        walkparams.s2pie = if IsFeatureImplemented(FEAT_S2PIE) then VTCR_EL2().S2PIE else '0';
    end;
    if IsFeatureImplemented(FEAT_S2PIE) then
        if !(HaveEL\(EL3\) && SCR_EL3().PIEn == '0' &&
            ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.s2pir = S2PIR_EL2() as S2PIRType;
        else
            walkparams.s2pir = Zeros{64} as S2PIRType;
        end;
    end;
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 != '1' then
        walkparams.assuredonly = VTCR_EL2().AssuredOnly;
    else
        walkparams.assuredonly = '0';
    end;
    walkparams.tl0 = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2().TL0 else '0';
    walkparams.tl1 = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2().TL1 else '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = VTCR_EL2().HAFT;
    else
        walkparams.haft = '0';
    end;
    if (IsFeatureImplemented(FEAT_HDBSS) && walkparams.hd == '1' &&
        (!HaveEL\(EL3\) || SCR_EL3().HDBSSEn == '1')) then
        walkparams.hdbss = VTCR_EL2().HDBSS;
    else
        walkparams.hdbss = '0';
    end;

    return walkparams;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_PAMax

```
// AArch64_PAMax()
// =====
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this PE

func AArch64_PAMax() => AddressSize
begin
    return ImpDefInt("Maximum Physical Address Size") as AddressSize;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_RLS2TTWParams

```
// AArch64_RLS2TTWParams()
// =====
// Gather walk parameters specific for Realm stage 2 translation

func AArch64_RLS2TTWParams(slaarch64 : boolean) => S2TTWParams
begin
    // Realm stage 2 walk parameters are similar to Non-secure
    var walkparams : S2TTWParams = AArch64\_NSS2TTWParams(slaarch64);
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
        IsSCTLR2EL2Enabled() then SCTLR2_EL2().EMEC else '0');
    return walkparams;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1DCacheEnabled

```
// AArch64_S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

func AArch64_S1DCacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime\_EL3 => return SCTLR_EL3().C == '1';
        when Regime\_EL2 => return SCTLR_EL2().C == '1';
        when Regime\_EL20 => return SCTLR_EL2().C == '1';
        when Regime\_EL10 => return SCTLR_EL1().C == '1';
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1DecodeTG0

```
// AArch64_S1DecodeTG0()
// =====
// Decode stage 1 granule size configuration bits TG0

func AArch64_S1DecodeTG0(tg0_in : bits(2)) => TGx
begin
    var tg0 : bits(2) = tg0_in;
    var tgx : TGx;

    if tg0 == '11' then
        tg0 = ImpDefBits{2}("TG0 encoded granule size");
    end;

    case tg0 of
        when '00' => tgx = TGx\_4KB;
        when '01' => tgx = TGx\_64KB;
        when '10' => tgx = TGx\_16KB;
    end;

    if !AArch64\_HaveS1TG(tgx) then
        case ImpDefBits{2}("TG0 encoded granule size") of
            when '00' => tgx = TGx\_4KB;
            when '01' => tgx = TGx\_64KB;
            when '10' => tgx = TGx\_16KB;
        end;
    end;

    return tgx;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1DecodeTG1

```
// AArch64_S1DecodeTG1()
// =====
// Decode stage 1 granule size configuration bits TG1

func AArch64_S1DecodeTG1(tg1_in : bits(2)) => TGx
begin
    var tg1 : bits(2) = tg1_in;
    var tgx : TGx;

    if tg1 == '00' then
        tg1 = ImpDefBits{2}("TG1 encoded granule size");
    end;

    case tg1 of
        when '10' =>    tgx = TGx_4KB;
        when '11' =>    tgx = TGx_64KB;
        when '01' =>    tgx = TGx_16KB;
    end;

    if !AArch64_HaveS1TG(tgx) then
        case ImpDefBits{2}("TG1 encoded granule size") of
            when '10' =>    tgx = TGx_4KB;
            when '11' =>    tgx = TGx_64KB;
            when '01' =>    tgx = TGx_16KB;
        end;
    end;

    return tgx;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1E0POEnabled

```
// AArch64_S1E0POEnabled()
// =====
// Determine whether stage 1 unprivileged permission overlay is enabled

func AArch64_S1E0POEnabled(regime : Regime, nv1 : bit) => boolean
begin
    assert HasUnprivileged(regime);

    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;
    end;

    case regime of
        when Regime_EL20 => return IsTCR2EL2Enabled() && TCR2_EL2().E0POE == '1';
        when Regime_EL10 => return IsTCR2EL1Enabled() && nv1 == '0' && TCR2_EL1().E0POE == '1';
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1EPD

```
// AArch64_S1EPD()
// =====
// Determine whether stage 1 translation table walk is allowed for the VA range

func AArch64_S1EPD(regime : Regime, va : bits(64)) => bit
begin
    assert HasUnprivileged(regime);
    let varange : VARange = AArch64_GetVARange(va);

    case regime of
        when Regime_EL20 => return (if varange == VARange_LOWER then TCR_EL2().EPD0
                                     else TCR_EL2().EPD1);
        when Regime_EL10 => return (if varange == VARange_LOWER then TCR_EL1().EPD0
                                     else TCR_EL1().EPD1);
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1Enabled

```
// AArch64_S1Enabled()
// =====
// Determine if stage 1 is enabled for the access type for this translation regime

func AArch64_S1Enabled(regime : Regime, acctype : AccessType) => boolean
begin
    if acctype == AccessType_TRBE && EffectiveTRBLIMITR_EL1_nVM() == '1' then
        return FALSE;
    end;
    if acctype == AccessType_SPE && EffectivePMBLIMITR_EL1_nVM() == '1' then
        return FALSE;
    end;
    case regime of
        when Regime_EL3 => return SCTL_EL3().M == '1';
        when Regime_EL2 => return SCTL_EL2().M == '1';
        when Regime_EL20 => return SCTL_EL2().M == '1';
        when Regime_EL10 => return ((EL2Enabled() || HCR_EL2().[DC,TGE] == '00') &&
                                   SCTL_EL1().M == '1');
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1ICacheEnabled

```
// AArch64_S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

func AArch64_S1ICacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime_EL3 => return SCTL_EL3().I == '1';
        when Regime_EL2 => return SCTL_EL2().I == '1';
        when Regime_EL20 => return SCTL_EL2().I == '1';
        when Regime_EL10 => return SCTL_EL1().I == '1';
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1MinTxSZ

```
// AArch64_S1MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

func AArch64_S1MinTxSZ(regime : Regime, walkparams : S1TTWParams) => integer
begin
    if IsFeatureImplemented(FEAT_LVA3) then
        if walkparams.d128 == '1' then
            if HasUnprivileged(regime) then
                return 9;
            else
                return 8;
            end;
        elsif walkparams.tgx == TGx_64KB || walkparams.ds == '1' then
            return 12;
        else
            return 16;
        end;
    end;
    if IsFeatureImplemented(FEAT_LVA) then
        if walkparams.tgx == TGx_64KB || walkparams.ds == '1' || walkparams.d128 == '1' then
            return 12;
        else
            return 16;
        end;
    end;
    return 16;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1POEnabled

```
// AArch64_S1POEnabled()
// =====
// Determine whether stage 1 privileged permission overlay is enabled

func AArch64_S1POEnabled(regime : Regime) => boolean
begin
    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;
    end;

    case regime of
        when Regime_EL3 => return TCR_EL3().POE == '1';
        when Regime_EL2 => return IsTCR2EL2Enabled() && TCR2_EL2().POE == '1';
        when Regime_EL20 => return IsTCR2EL2Enabled() && TCR2_EL2().POE == '1';
        when Regime_EL10 => return IsTCR2EL1Enabled() && TCR2_EL1().POE == '1';
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1POR

```
// AArch64_S1POR()
// =====
// Identify stage 1 permissions overlay register for the acting translation regime

func AArch64_S1POR(regime : Regime) => S1PORType
begin
    if (HaveEL(EL3) && SCR_EL3().PIEn == '0' && regime != Regime_EL3 &&
        ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
        return Zeros{64} as S1PORType;
    end;

    case regime of
        when Regime_EL3 => return POR_EL3() as S1PORType;
        when Regime_EL2 => return POR_EL2() as S1PORType;
        when Regime_EL20 => return POR_EL2() as S1PORType;
        when Regime_EL10 => return POR_EL1() as S1PORType;
    end;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1TTBR

```
// AArch64_S1TTBR()
// =====
// Identify stage 1 table base register for the acting translation regime

func AArch64_S1TTBR(regime : Regime, va : bits(64)) => bits(128)
begin
    let varange : VARange = AArch64_GetVARange(va);

    case regime of
        when Regime_EL3 => return ZeroExtend{128}(TTBR0_EL3());
        when Regime_EL2 => return ZeroExtend{128}(TTBR0_EL2());
        when Regime_EL20 =>
            if varange == VARange_LOWER then
                return ZeroExtend{128}(TTBR0_EL2());
            else
                return ZeroExtend{128}(TTBR1_EL2());
            end;
        when Regime_EL10 =>
            if varange == VARange_LOWER then
                return ZeroExtend{128}(TTBR0_EL1());
            else
                return ZeroExtend{128}(TTBR1_EL1());
            end;
    end;
end;
```



```

// AArch64_S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

func AArch64_S1TTWParamsEL10(e1 : bits(2), varange : VARange) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    if IsFeatureImplemented(FEAT_D128) && IsTCR2EL1Enabled() then
        walkparams.d128 = TCR2_EL1().D128;
    else
        walkparams.d128 = '0';
    end;
    let nvs : bits(3) = EffectiveHCR_EL2_NVx();
    walkparams.nv1 = nvs[1];

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL1() as MAIRType;
    end;
    walkparams.aie = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL1Enabled() then TCR2_EL1().AIE
        else '0');
    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) &&
            IsTCR2EL1Enabled() then TCR2_EL1().PIE else '0');
    end;
    if IsFeatureImplemented(FEAT_S1PIE) then
        if ! (HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
            ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.pir = PIR_EL1() as S1PIRType;
            if walkparams.nv1 == '1' then
                walkparams.pire0 = Zeros{64} as S1PIRType;
            else
                walkparams.pire0 = PIRE0_EL1() as S1PIRType;
            end;
        else
            walkparams.pir = Zeros{64} as S1PIRType;
            walkparams.pire0 = Zeros{64} as S1PIRType;
        end;
    end;

    if varange == VARange_LOWER then
        walkparams.tgx = AArch64_S1DecodeTGO(TCR_EL1().TG0);
        walkparams.txsz = TCR_EL1().TOSZ;
        walkparams.irgn = TCR_EL1().IRGN0;
        walkparams.orgn = TCR_EL1().ORGN0;
        walkparams.sh = TCR_EL1().SH0;
        walkparams.tbi = TCR_EL1().TBI0;

        walkparams.nfd = if IsFeatureImplemented(FEAT_SVE) then TCR_EL1().NFD0 else '0';
        walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1().TBID0 else '0';
        walkparams.eopd = if IsFeatureImplemented(FEAT_EOPD) then TCR_EL1().EOPD0 else '0';
        walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL1().HPD0 else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1'; end;
            if walkparams.pie == '1' then walkparams.hpd = '1'; end;
            if (AArch64_S1POEnabled(Regime_EL10) ||
                AArch64_S1EOPOEnabled(Regime_EL10, walkparams.nv1)) then walkparams.hpd = '1';
            end;
        end;
        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
            IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
            walkparams.mtx = TCR_EL1().MTX0;
        else
            walkparams.mtx = '0';
        end;
        walkparams.sk1 = if walkparams.d128 == '1' then TTBR0_EL1().SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1().DisCH0 else '0';
        if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() then
            walkparams.fng = TCR2_EL1().FNG0;
        else
            walkparams.fng = '0';
        end;
        if IsFeatureImplemented(FEAT_THE) && IsTCR2EL1Enabled() then
            walkparams.fngna = TCR2_EL1().FNGNA0;
        else
            walkparams.fngna = '0';
        end;
    else
        walkparams.tgx = AArch64_S1DecodeTG1(TCR_EL1().TG1);
        walkparams.txsz = TCR_EL1().T1SZ;
        walkparams.irgn = TCR_EL1().IRGN1;
        walkparams.orgn = TCR_EL1().ORGN1;
        walkparams.sh = TCR_EL1().SH1;
        walkparams.tbi = TCR_EL1().TBI1;

        walkparams.nfd = if IsFeatureImplemented(FEAT_SVE) then TCR_EL1().NFD1 else '0';

```



```

walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1().TBID1 else '0';
walkparams.eOpd = if IsFeatureImplemented(FEAT_EOPD) then TCR_EL1().EOPD1 else '0';
walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL1().HPD1 else '0';
if walkparams.hpd == '0' then
    if walkparams.aie == '1' then walkparams.hpd = '1'; end;
    if walkparams.pie == '1' then walkparams.hpd = '1'; end;
    if (AArch64_S1POEnabled(Regime_EL10) ||
        AArch64_S1E0POEnabled(Regime_EL10, walkparams.nv1)) then walkparams.hpd = '1';
    end;
end;
if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
    IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
    walkparams.mtx = TCR_EL1().MTX1;
else
    walkparams.mtx = '0';
end;
walkparams.sk1 = if walkparams.d128 == '1' then TTBRI_EL1().SKL else '00';
walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1().DisCH1 else '0';
if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() then
    walkparams.fng = TCR2_EL1().FNG1;
else
    walkparams.fng = '0';
end;
if IsFeatureImplemented(FEAT_THE) && IsTCR2EL1Enabled() then
    walkparams.fngna = TCR2_EL1().FNGNA1;
else
    walkparams.fngna = '0';
end;
end;

walkparams.mair = MAIR_EL1() as MAIRType;
walkparams.wxn = SCTLR_EL1().WXN;
walkparams.ps = TCR_EL1().IPS;
walkparams.ee = SCTLR_EL1().EE;
if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2))) then
    walkparams.sif = SCR_EL3().SIF;
else
    walkparams.sif = '0';
end;

if EL2Enabled() then
    walkparams.dc = HCR_EL2().DC;
    walkparams.dct = if IsFeatureImplemented(FEAT_MTE2) then HCR_EL2().DCT else '0';
end;

if IsFeatureImplemented(FEAT_LSMAOC) then
    walkparams.ntlsmid = SCTLR_EL1().nTlSMID;
else
    walkparams.ntlsmid = '1';
end;

walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW) then SCTLR_EL1().CMOW else '0';
walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then TCR_EL1().HA else '0';
if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
    walkparams.hd = TCR_EL1().HD;
else
    walkparams.hd = '0';
end;
if (walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) &&
    walkparams.d128 == '0') then
    walkparams.ds = TCR_EL1().DS;
else
    walkparams.ds = '0';
end;
if IsFeatureImplemented(FEAT_PAN3) then
    walkparams.epan = if walkparams.pie == '0' then SCTLR_EL1().EPAN else '1';
else
    walkparams.epan = '0';
end;
if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL1Enabled() then
    walkparams.pnch = TCR2_EL1().PnCH;
else
    walkparams.pnch = '0';
end;
if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL1Enabled() then
    walkparams.haft = TCR2_EL1().HAFT;
else
    walkparams.haft = '0';
end;
walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
    IsSCTLR2EL2Enabled() then SCTLR2_EL2().EMEC else '0');

return walkparams;
end;

```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1TTWParamsEL2

```
// AArch64_S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

func AArch64_S1TTWParamsEL2(ss : SecurityState) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    walkparams.tgx = AArch64_S1DecodeTGO(TCR_EL2().TG0);
    walkparams.txsx = TCR_EL2().TOSZ;
    walkparams.ps = TCR_EL2().PS;
    walkparams.irgn = TCR_EL2().IRGN0;
    walkparams.orgn = TCR_EL2().ORGN0;
    walkparams.sh = TCR_EL2().SH0;
    walkparams.tbi = TCR_EL2().TBI;
    walkparams.mair = MAIR_EL2() as MAIRType;

    walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) && IsTCR2EL2Enabled() then TCR2_EL2().PIE
                     else '0');

    if IsFeatureImplemented(FEAT_S1PIE) then
        if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
            ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.pir = PIR_EL2() as S1PIRType;
        else
            walkparams.pir = Zeros{64} as S1PIRType;
        end;
    end;

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL2() as MAIRType;
    end;

    walkparams.aie = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL2Enabled() then TCR2_EL2().AIE
                     else '0');
    walkparams.wxn = SCTLR_EL2().WXN;
    walkparams.ee = SCTLR_EL2().EE;
    if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2))) then
        walkparams.sif = SCR_EL3().SIF;
    else
        walkparams.sif = '0';
    end;

    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2().TBID else '0';
    walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL2().HPD else '0';
    if walkparams.hpd == '0' then
        if walkparams.aie == '1' then walkparams.hpd = '1'; end;
        if walkparams.pie == '1' then walkparams.hpd = '1'; end;
        if AArch64_S1POEnabled(Regime_EL2) then walkparams.hpd = '1'; end;
    end;

    walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then TCR_EL2().HA else '0';
    if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
        walkparams.hd = TCR_EL2().HD;
    else
        walkparams.hd = '0';
    end;

    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = TCR_EL2().DS;
    else
        walkparams.ds = '0';
    end;

    if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
        IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
        walkparams.mtx = TCR_EL2().MTX;
    else
        walkparams.mtx = '0';
    end;

    walkparams.pnch = (if IsFeatureImplemented(FEAT_THE) && IsTCR2EL2Enabled() then TCR2_EL2().PnCH
                     else '0');
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL2Enabled() then
        walkparams.haft = TCR2_EL2().HAFT;
    else
        walkparams.haft = '0';
    end;

    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
                       IsSCTLR2EL2Enabled() then SCTLR2_EL2().EMEC else '0');
    if IsFeatureImplemented(FEAT_MEC) && ss == SS_Realm && IsTCR2EL2Enabled() then
        walkparams.amec = TCR2_EL2().AMEC0;
    else
        walkparams.amec = '0';
    end;

    return walkparams;
end;
```



```

// AArch64_S1TTWParamsEL20()
// =====
// Gather stage 1 translation table walk parameters for EL2&0 regime

func AArch64_S1TTWParamsEL20(el : bits(2), ss : SecurityState, varange : VARange) => S1TTWParams
begin
    var walkparams : S1TTWParams;

    if IsFeatureImplemented(FEAT_D128) && IsTCR2EL2Enabled() then
        walkparams.d128 = TCR2_EL2().D128;
    else
        walkparams.d128 = '0';
    end;

    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) &&
IsTCR2EL2Enabled() then TCR2_EL2().PIE else '0');
    end;

    if IsFeatureImplemented(FEAT_S1PIE) then
        if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
            ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.pir = PIR_EL2() as S1PIRType;
            walkparams.pire0 = PIRE0_EL2() as S1PIRType;
        else
            walkparams.pir = Zeros{64} as S1PIRType;
            walkparams.pire0 = Zeros{64} as S1PIRType;
        end;
    end;

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL2() as MAIRType;
    end;

    walkparams.aie = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL2Enabled() then TCR2_EL2().AIE
        else '0');

    if varange == VARange\_LOWER then
        walkparams.tgx = AArch64\_S1DecodeTG0(TCR_EL2().TG0);
        walkparams.txsz = TCR_EL2().T0SZ;
        walkparams.irgn = TCR_EL2().IRGN0;
        walkparams.orgn = TCR_EL2().ORGN0;
        walkparams.sh = TCR_EL2().SH0;
        walkparams.tbi = TCR_EL2().TBI0;

        walkparams.nfd = if IsFeatureImplemented(FEAT_SVE) then TCR_EL2().NFD0 else '0';
        walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2().TBID0 else '0';
        walkparams.eopd = if IsFeatureImplemented(FEAT_EOPD) then TCR_EL2().EOPD0 else '0';
        walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL2().HPD0 else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1'; end;
            if walkparams.pie == '1' then walkparams.hpd = '1'; end;
            if AArch64\_S1POEnabled(Regime\_EL20) || AArch64\_S1E0POEnabled(Regime\_EL20, '0') then
                walkparams.hpd = '1';
            end;
        end;

        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
            IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
            walkparams.mtx = TCR_EL2().MTX0;
        else
            walkparams.mtx = '0';
        end;

        walkparams.sk1 = if walkparams.d128 == '1' then TTBR0_EL2().SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2().DisCH0 else '0';
        if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled() then
            walkparams.fng = TCR2_EL2().FNG0;
        else
            walkparams.fng = '0';
        end;
    else
        walkparams.tgx = AArch64\_S1DecodeTG1(TCR_EL2().TG1);
        walkparams.txsz = TCR_EL2().T1SZ;
        walkparams.irgn = TCR_EL2().IRGN1;
        walkparams.orgn = TCR_EL2().ORGN1;
        walkparams.sh = TCR_EL2().SH1;
        walkparams.tbi = TCR_EL2().TBI1;

        walkparams.nfd = if IsFeatureImplemented(FEAT_SVE) then TCR_EL2().NFD1 else '0';
        walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2().TBID1 else '0';
        walkparams.eopd = if IsFeatureImplemented(FEAT_EOPD) then TCR_EL2().EOPD1 else '0';
        walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL2().HPD1 else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1'; end;
            if walkparams.pie == '1' then walkparams.hpd = '1'; end;
            if AArch64\_S1POEnabled(Regime\_EL20) || AArch64\_S1E0POEnabled(Regime\_EL20, '0') then
                walkparams.hpd = '1';
            end;
        end;

        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
            IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then

```

```

        walkparams.mtx = TCR_EL2().MTX1;
    else
        walkparams.mtx = '0';
    end;
    walkparams.sk1 = if walkparams.d128 == '1' then TTBR1_EL2().SKL else '00';
    walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2().DisCH1 else '0';
    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled\(\) then
        walkparams.fng = TCR2_EL2().FNG1;
    else
        walkparams.fng = '0';
    end;
end;

walkparams.mair = MAIR_EL2() as MAIRType;
walkparams.wxn = SCTLR_EL2().WXN;
walkparams.ps = TCR_EL2().IPS;
walkparams.ee = SCTLR_EL2().EE;
if (HaveEL\(EL3\)) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2)) then
    walkparams.sif = SCR_EL3().SIF;
else
    walkparams.sif = '0';
end;

if IsFeatureImplemented(FEAT_LSMAOC) then
    walkparams.ntlsmd = SCTLR_EL2().nTLSMD;
else
    walkparams.ntlsmd = '1';
end;

walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW) then SCTLR_EL2().CMOW else '0';
walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then TCR_EL2().HA else '0';
if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
    walkparams.hd = TCR_EL2().HD;
else
    walkparams.hd = '0';
end;
if (walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && IsFeatureImplemented(FEAT_LPA2) &&
    walkparams.d128 == '0') then
    walkparams.ds = TCR_EL2().DS;
else
    walkparams.ds = '0';
end;
if IsFeatureImplemented(FEAT_PAN3) then
    walkparams.epan = if walkparams.pie == '0' then SCTLR_EL2().EPAN else '1';
else
    walkparams.epan = '0';
end;
if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL2Enabled\(\) then
    walkparams.pnch = TCR2_EL2().PnCH;
else
    walkparams.pnch = '0';
end;
if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL2Enabled\(\) then
    walkparams.haft = TCR2_EL2().HAFT;
else
    walkparams.haft = '0';
end;
walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) && IsSCTLR2EL2Enabled\(\)
    then SCTLR2_EL2().EMEC else '0');
if IsFeatureImplemented(FEAT_MEC) && ss == SS\_Realm && IsTCR2EL2Enabled\(\) then
    walkparams.amec = if varange == VARange\_LOWER then TCR2_EL2().AMECO else TCR2_EL2().AMEC1;
else
    walkparams.amec = '0';
end;

return walkparams;
end;

```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S1TTWParamsEL3

```
// AArch64_S1TTWParamsEL3()
// =====
// Gather stage 1 translation table walk parameters for EL3 regime

func AArch64_S1TTWParamsEL3() => S1TTWParams
begin
    var walkparams : S1TTWParams;

    walkparams.tgx = AArch64\_S1DecodeTGO(TCR_EL3().TG0);
    walkparams.txsz = TCR_EL3().TOSZ;
    walkparams.ps = TCR_EL3().PS;
    walkparams.irgn = TCR_EL3().IRGN0;
    walkparams.orgn = TCR_EL3().ORGN0;
    walkparams.sh = TCR_EL3().SH0;
    walkparams.tbi = TCR_EL3().TBI;
    walkparams.mair = MAIR_EL3() as MAIRType;
    walkparams.d128 = if IsFeatureImplemented(FEAT_D128) then TCR_EL3().D128 else '0';
    walkparams.sk1 = if walkparams.d128 == '1' then TTBR0_EL3().SKL else '00';
    walkparams.disch = if walkparams.d128 == '1' then TCR_EL3().DisCH0 else '0';

    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = if IsFeatureImplemented(FEAT_S1PIE) then TCR_EL3().PIE else '0';
    end;
    if IsFeatureImplemented(FEAT_S1PIE) then
        walkparams.pir = PIR_EL3() as S1PIRType;
    end;

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL3() as MAIRType;
    end;
    walkparams.aie = if IsFeatureImplemented(FEAT_AIE) then TCR_EL3().AIE else '0';
    walkparams.wxn = SCTLR_EL3().WXN;
    walkparams.ee = SCTLR_EL3().EE;
    walkparams.sif = (if !IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2)
        then SCR_EL3().SIF else '0');

    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL3().TBID else '0';
    walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL3().HPD else '0';
    if walkparams.hpd == '0' then
        if walkparams.aie == '1' then walkparams.hpd = '1'; end;
        if walkparams.pie == '1' then walkparams.hpd = '1'; end;
        if AArch64\_S1POEnabled\(Regime\_EL3\) then walkparams.hpd = '1'; end;
    end;
    walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then TCR_EL3().HA else '0';
    if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
        walkparams.hd = TCR_EL3().HD;
    else
        walkparams.hd = '0';
    end;
    if (walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && IsFeatureImplemented(FEAT_LPA2) &&
        walkparams.d128 == '0') then
        walkparams.ds = TCR_EL3().DS;
    else
        walkparams.ds = '0';
    end;
    if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
        IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
        walkparams.mtx = TCR_EL3().MTX;
    else
        walkparams.mtx = '0';
    end;
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' then
        walkparams.pnch = TCR_EL3().PnCH;
    else
        walkparams.pnch = '0';
    end;
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = TCR_EL3().HAFT;
    else
        walkparams.haft = '0';
    end;
    walkparams.emec = if IsFeatureImplemented(FEAT_MEC) then SCTLR2_EL3().EMEC else '0';

    return walkparams;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S2DecodeTG0

```
// AArch64_S2DecodeTG0()
// =====
// Decode stage 2 granule size configuration bits TG0

func AArch64_S2DecodeTG0(tg0_in : bits(2)) => TGx
begin
    var tg0 : bits(2) = tg0_in;
    var tgx : TGx;

    if tg0 == '11' then
        tg0 = ImpDefBits{2}("TG0 encoded granule size");
    end;

    case tg0 of
        when '00' => tgx = TGx_4KB;
        when '01' => tgx = TGx_64KB;
        when '10' => tgx = TGx_16KB;
    end;

    if !AArch64_HaveS2TG(tgx) then
        case ImpDefBits{2}("TG0 encoded granule size") of
            when '00' => tgx = TGx_4KB;
            when '01' => tgx = TGx_64KB;
            when '10' => tgx = TGx_16KB;
        end;
    end;

    return tgx;
end;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64_S2MinTxSZ

```
// AArch64_S2MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2

func AArch64_S2MinTxSZ(walkparams : S2TTWParams, slaarch64 : boolean) => integer
begin
    var ips : integer;

    if AArch64_PAMax() == 56 then
        if walkparams.d128 == '1' then
            ips = 56;
        elsif walkparams.tgx == TGx_64KB || walkparams.ds == '1' then
            ips = 52;
        else
            ips = 48;
        end;
    elsif AArch64_PAMax() == 52 then
        if walkparams.tgx == TGx_64KB || walkparams.ds == '1' then
            ips = 52;
        else
            ips = 48;
        end;
    else
        ips = AArch64_PAMax();
    end;

    var min_txsz : integer = 64 - ips;
    if !slaarch64 then
        // EL1 is AArch32
        min_txsz = Min(min_txsz, 24);
    end;

    return min_txsz;
end;
```



```

// AArch64_SS2TTWParams()
// =====
// Gather walk parameters specific for secure stage 2 translation

func AArch64_SS2TTWParams(ipaspace : PASpace, slaarch64 : boolean) => S2TTWParams
begin
    var walkparams : S2TTWParams;

    walkparams.dl28 = if IsFeatureImplemented(FEAT_D128) then VTCR_EL2().D128 else '0';
    if ipaspace == PAS\_Secure then
        walkparams.tgx = AArch64\_S2DecodeTG0(VTCR_EL2().TG0);
        walkparams.txs = VTCR_EL2().TOSZ;
        if walkparams.dl28 == '1' then
            walkparams.sk1 = VSTBR_EL2().SKL;
        else
            walkparams.sl0 = VTCR_EL2().SL0;
        end;
        if walkparams.tgx == TGx\_4KB && IsFeatureImplemented(FEAT_LPA2) then
            walkparams.sl2 = VTCR_EL2().SL2 AND VTCR_EL2().DS;
        else
            walkparams.sl2 = '0';
        end;
    elsif ipaspace == PAS\_NonSecure then
        walkparams.tgx = AArch64\_S2DecodeTG0(VTCR_EL2().TG0);
        walkparams.txs = VTCR_EL2().TOSZ;
        if walkparams.dl28 == '1' then
            walkparams.sk1 = VTTBR_EL2().SKL;
        else
            walkparams.sl0 = VTCR_EL2().SL0;
        end;
        if walkparams.tgx == TGx\_4KB && IsFeatureImplemented(FEAT_LPA2) then
            walkparams.sl2 = VTCR_EL2().SL2 AND VTCR_EL2().DS;
        else
            walkparams.sl2 = '0';
        end;
    else
        unreachable;
    end;

    walkparams.sw = VTCR_EL2().SW;
    walkparams.nsw = VTCR_EL2().NSW;
    walkparams.sa = VTCR_EL2().SA;
    walkparams.nsa = VTCR_EL2().NSA;
    walkparams.vm = HCR_EL2().VM OR HCR_EL2().DC;
    walkparams.ps = VTCR_EL2().PS;
    walkparams.irgn = VTCR_EL2().IRGN0;
    walkparams.orgn = VTCR_EL2().ORGN0;
    walkparams.sh = VTCR_EL2().SH0;
    walkparams.ee = SCTLR_EL2().EE;

    walkparams.ptw = if HCR_EL2().TGE == '0' then HCR_EL2().PTW else '0';
    walkparams.fwb = if IsFeatureImplemented(FEAT_S2FWB) then HCR_EL2().FWB else '0';
    walkparams.ha = if IsFeatureImplemented(FEAT_HAF) then VTCR_EL2().HA else '0';
    if IsFeatureImplemented(FEAT_HAFDBS) && walkparams.ha == '1' then
        walkparams.hd = VTCR_EL2().HD;
    else
        walkparams.hd = '0';
    end;
    if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = VTCR_EL2().DS;
    else
        walkparams.ds = '0';
    end;
    walkparams.cmw = (if IsFeatureImplemented(FEAT_CMOW) && IsHCRXEL2Enabled() then HCRX_EL2().CMOW
        else '0');
    if walkparams.dl28 == '1' then
        walkparams.s2pie = '1';
    else
        walkparams.s2pie = if IsFeatureImplemented(FEAT_S2PIE) then VTCR_EL2().S2PIE else '0';
    end;
    if IsFeatureImplemented(FEAT_S2PIE) then
        if !(HaveEL(EL3) && SCR_EL3().PIEn == '0' &&
            ImpDefBool("SCR_EL3.PIEn forces PIR_ELx or POR_ELx to zero")) then
            walkparams.s2pir = S2PIR_EL2() as S2PIRType;
        else
            walkparams.s2pir = Zeros{64} as S2PIRType;
        end;
    end;
    if IsFeatureImplemented(FEAT_THE) && walkparams.dl28 != '1' then
        walkparams.assuredonly = VTCR_EL2().AssuredOnly;
    else
        walkparams.assuredonly = '0';
    end;
    walkparams.tl0 = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2().TL0 else '0';
    walkparams.tl1 = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2().TL1 else '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = VTCR_EL2().HAFT;
    else
        walkparams.haft = '0';
    end;
end

```

```

end;
walkparams.emec = '0';
if (IsFeatureImplemented(FEAT_HDBSS) && walkparams.hd == '1' &&
    (!HaveEL(EL3) || SCR_EL3().HDBSSen == '1')) then
    walkparams.hdbss = VTCR_EL2().HDBSS;
else
    walkparams.hdbss = '0';
end;

return walkparams;
end;

```

Library pseudocode for aarch64/translation/vmsa_walkparams/S2DCacheEnabled

```

// S2DCacheEnabled()
// =====
// Returns TRUE if Stage 2 Data access cacheability is enabled

func S2DCacheEnabled() => boolean
begin
    return HCR_EL2().CD == '0';
end;

```

Library pseudocode for parameters

```

constant integer{} NUM_ASIDBITS = 16;
constant integer{} NUM_VMIDBITS = 16;

```

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```

// ClearStickyErrors()
// =====

func ClearStickyErrors()
begin
    EDSCR().TXU = '0';           // Clear TX underrun flag
    EDSCR().RXO = '0';           // Clear RX overrun flag

    if Halted() then             // in Debug state
        EDSCR().ITO = '0';       // Clear ITR overrun flag
    end;

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if (Halted() && EDSCR().ITE == '0' &&
        ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO)) then
        return;
    end;
    EDSCR().ERR = '0';           // Clear cumulative error flag

    return;
end;

```

Library pseudocode for shared/debug/DebugTarget/DebugTarget

```

// DebugTarget()
// =====
// Returns the debug exception target Exception level

func DebugTarget() => bits(2)
begin
    let ss : SecurityState = CurrentSecurityState();
    return DebugTargetFrom(ss);
end;

```

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =====

func DebugTargetFrom(from_state : SecurityState) => bits(2)
begin
    var route_to_el2 : boolean;
    if HaveEL\(EL2\) && (from_state != SS\_Secure ||
        (IsFeatureImplemented(FEAT_SEL2) && (!HaveEL\(EL3\) || SCR_EL3().EEL2 == '1'))) then
        if ELUsingAArch32\(EL2\) then
            route_to_el2 = (HDCR().TDE == '1' || HCR().TGE == '1');
        else
            route_to_el2 = (MDCR_EL2().TDE == '1' || HCR_EL2().TGE == '1');
        end;
    else
        route_to_el2 = FALSE;
    end;

    var target : bits(2);
    if route_to_el2 then
        target = EL2;
    elsif HaveEL\(EL3\) && !HaveAArch64\(\) && from_state == SS\_Secure then
        target = EL3;
    else
        target = EL1;
    end;

    return target;
end;
```

Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// =====
// Returns the state of the OS Double Lock.
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

func DoubleLockStatus() => boolean
begin
    if !IsFeatureImplemented(FEAT_DoubleLock) then
        return FALSE;
    end;
    if ELUsingAArch32\(EL1\) then
        return DBGOSDLR().DLK == '1' && DBGPRCR().CORENPDRQ == '0' && !Halted\(\);
    else
        return OSDLR_EL1().DLK == '1' && DBGPRCR_EL1().CORENPDRQ == '0' && !Halted\(\);
    end;
end;
```

Library pseudocode for shared/debug/OSLockStatus/OSLockStatus

```
// OSLockStatus()
// =====
// Returns the state of the OS Lock.

readonly func OSLockStatus() => boolean
begin
    return (if ELUsingAArch32\(EL1\) then DBGOSLSR().OSLK else OSLSR_EL1().OSLK) == '1';
end;
```

Library pseudocode for shared/debug/SoftwareLockStatus/Component

```
// Component
// =====
// Component Types.

type Component of enumeration {
    Component_AMU,
    Component_ETE,
    Component_TRBE,
    Component_RAS,
    Component_GIC,
    Component_PMU,
    Component_Debug,
    Component_CTI
};
```

Library pseudocode for shared/debug/SoftwareLockStatus/GetAccessComponent

```
// GetAccessComponent()
// =====
// Determines the component by decoding the physical address in the address descriptor.

impdef func GetAccessComponent(addrdesc : AddressDescriptor) => Component
begin
    return Component\_Debug;
end;
```

Library pseudocode for shared/debug/SoftwareLockStatus/SoftwareLockStatus

```
// SoftwareLockStatus()
// =====
// Returns the state of the Software Lock.

func SoftwareLockStatus(addrdesc : AddressDescriptor) => boolean
begin
    let component : Component = GetAccessComponent(addrdesc);
    if !HaveSoftwareLock(component) then
        return FALSE;
    end;
    case component of
        when Component\_ETE => return TRCLSR().SLK == '1';
        when Component\_Debug => return EDLSR().SLK == '1';
        when Component\_PMU => return PMLSR().SLK == '1';
        when Component\_CTI => return CTILSR().SLK == '1';
        otherwise => return FALSE;
    end;
end;
```

Library pseudocode for shared/debug/amu/IsG1ActivityMonitorImplemented

```
// IsG1ActivityMonitorImplemented()
// =====
// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.

impdef func IsG1ActivityMonitorImplemented(i : integer) => boolean
begin
  case i of
    when 0 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 0"));
    when 1 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 1"));
    when 2 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 2"));
    when 3 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 3"));
    when 4 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 4"));
    when 5 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 5"));
    when 6 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 6"));
    when 7 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 7"));
    when 8 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 8"));
    when 9 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 9"));
    when 10 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 10"));
    when 11 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 11"));
    when 12 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 12"));
    when 13 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 13"));
    when 14 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 14"));
    when 15 =>
      return (ImpDefBool(
        "G1 activity monitor is implemented for counter 15"));
    otherwise =>
      return FALSE;
  end;
end;
```

Library pseudocode for shared/debug/amu/IsG1ActivityMonitorOffsetImplemented

```
// IsG1ActivityMonitorOffsetImplemented()
// =====
// Returns TRUE if a G1 activity monitor offset is implemented for the counter,
// and FALSE otherwise.

impdef func IsG1ActivityMonitorOffsetImplemented(i : integer) => boolean
begin
  case i of
    when 0 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 0"));
    when 1 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 1"));
    when 2 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 2"));
    when 3 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 3"));
    when 4 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 4"));
    when 5 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 5"));
    when 6 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 6"));
    when 7 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 7"));
    when 8 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 8"));
    when 9 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 9"));
    when 10 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 10"));
    when 11 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 11"));
    when 12 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 12"));
    when 13 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 13"));
    when 14 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 14"));
    when 15 =>
      return (ImpDefBool(
        "G1 activity monitor offset is implemented for counter 15"));
    otherwise =>
      return FALSE;
  end;
end;
```

Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External DBGVR<n>_EL1,
// DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1 registers, and, from Armv8.2, the
// OSLAR_EL1 register is allowed for the access. Returns FALSE otherwise.

func AllowExternalDebugAccess(addrdesc : AddressDescriptor) => boolean
begin
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3().[EDADE,EDAD] of
            when '00' => return TRUE;
            when '01' => return addrdesc.paddress.paspace IN {PAS\_Root, PAS\_Secure};
            when '10' => return addrdesc.paddress.paspace IN {PAS\_Root, PAS\_Realm};
            when '11' => return addrdesc.paddress.paspace == PAS\_Root;
        end;
    end;

    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if addrdesc.paddress.paspace == PAS\_Secure then return TRUE; end;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE; end;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE; end;
    end;

    if HaveEL\(EL3\) then
        let EDAD_bit : bit = if ELUsingAArch32\(EL3\) then SDCR().EDAD else MDCR_EL3().EDAD;
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
    end;
end;
```

Library pseudocode for shared/debug/authentication/AllowExternalPMSSAccess

```
// AllowExternalPMSSAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU Snapshot
// registers is allowed for the given Security state, FALSE otherwise.

func AllowExternalPMSSAccess(addrdesc : AddressDescriptor) => boolean
begin
    assert IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64();
    // FEAT_Debugv8p4 is always implemented when FEAT_PMUv3_SS is implemented.
    assert IsFeatureImplemented(FEAT_Debugv8p4);

    // The access may also be subject to the OS Double Lock, power-down, etc.
    var epmsad : bits(2) = if HaveEL\(EL3\) then MDCR_EL3().EPMSSAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && epmsad IN {'01','10'} then
        (-, epmsad) = ConstrainUnpredictableBits(2)(Unpredictable\_RESEPMSSAD);
        // The value returned by ConstrainUnpredictableBits() must be a
        // non-reserved value
        assert epmsad IN {'00','11'};
    end;

    case epmsad of
        when '00' =>
            if IsFeatureImplemented(FEAT_RME) then
                return addrdesc.paddress.paspace == PAS\_Root;
            else
                return addrdesc.paddress.paspace == PAS\_Secure;
            end;
        when '01' =>
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS\_Root, PAS\_Realm};
        when '10' =>
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS\_Root, PAS\_Secure};
        when '11' =>
            return TRUE;
    end;
end;
```

Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

func AllowExternalPMUAccess(addrdesc : AddressDescriptor) => boolean
begin
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3().[EPMADE, EPMA] of
            when '00' => return TRUE;
            when '01' => return addrdesc.paddress.paspace IN {PAS\_Root, PAS\_Secure};
            when '10' => return addrdesc.paddress.paspace IN {PAS\_Root, PAS\_Realm};
            when '11' => return addrdesc.paddress.paspace == PAS\_Root;
        end;
    end;

    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if addrdesc.paddress.paspace == PAS\_Secure then return TRUE; end;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE; end;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE; end;
    end;

    if HaveEL(EL3) then
        let EPMA_bit : bit = if ELUsingAArch32(EL3) then SDCR().EPMA else MDCR_EL3().EPMA;
        return EPMA_bit == '0';
    else
        return NonSecureOnlyImplementation();
    end;
end;
```

Library pseudocode for shared/debug/authentication/AllowExternalTraceAccess

```
// AllowExternalTraceAccess()
// =====
// Returns TRUE if an external Trace access to the Trace registers is allowed for the
// given PAS, FALSE otherwise.

func AllowExternalTraceAccess(addrdesc : AddressDescriptor) => boolean
begin
    // The access may also be subject to OS lock, power-down, etc.
    if !IsFeatureImplemented(FEAT_TRBE) then return TRUE; end;
    assert IsFeatureImplemented(FEAT_Debugv8p4);
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3().[ETADE, ETAD] of
            when '00' => return TRUE;
            when '01' => return addrdesc.paddress.paspace IN {PAS\_Root, PAS\_Secure};
            when '10' => return addrdesc.paddress.paspace IN {PAS\_Root, PAS\_Realm};
            when '11' => return addrdesc.paddress.paspace == PAS\_Root;
        end;
    end;

    if addrdesc.paddress.paspace == PAS\_Secure then return TRUE; end;
    if HaveEL(EL3) then
        // External Trace access is not supported for EL3 using AArch32
        assert !ELUsingAArch32(EL3);
        return MDCR_EL3().ETAD == '0';
    else
        return NonSecureOnlyImplementation();
    end;
end;
```


Library pseudocode for shared/debug/authentication/AllowRASMemoryMappedAccess

```
// AllowRASMemoryMappedAccess()
// =====
// Returns TRUE if read and/or write access to RAS memory-mapped registers is
// permitted for the given access.

func AllowRASMemoryMappedAccess(addrdesc : AddressDescriptor, iswrite : boolean) => boolean
begin
    // An access from the most secure implemented PAS is always permitted.
    if IsMostSecureAccess(addrdesc) then
        return TRUE;
    end;

    if IsFeatureImplemented(FEAT_RASSA_ACR) && IsFeatureImplemented(FEAT_RASSAv2) then
        var ra : bits(2);
        let implements_rl_and_s : boolean = (IsFeatureImplemented(FEAT_RME) &&
            ImpDefBool(
                "the error record group allows configuration of Secure and Realm register accesses"));
        if IsAccessRealm(addrdesc) then
            ra = if implements_rl_and_s then ERRACR().RLRA else ERRACR().NSRA;
        elseif IsAccessSecure(addrdesc) then
            ra = if implements_rl_and_s then ERRACR().SRA else ERRACR().NSRA;
        elseif IsAccessNonSecure(addrdesc) then
            ra = ERRACR().NSRA;
        else
            unreachable;
        end;
        // '10' is reserved
        if ra == '10' then (-, ra) = ConstrainUnpredictableBits{2}(Unpredictable\_RES\_RA); end;
        // Interpret configured permissions to a boolean for the requested access
        case ra of
            when '00' => return FALSE;           // RAZ/WI => no read or write
            when '01' => return !iswrite;        // RO => reads allowed, writes not
            when '11' => return TRUE;             // RW => both allowed
            otherwise => unreachable;
        end;
    end;
    return ImpDefBool("error records are accessible to all PAS");
end;
```

Library pseudocode for shared/debug/authentication/Debug

```
// Debug authentication signals
// =====

var DBGEN : Signal;
var NIDEN : Signal;
var SPIDEN : Signal;
var SPNIDEN : Signal;
var RLPIDEN : Signal;
var RTPIDEN : Signal;
```

Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

readonly func ExternalInvasiveDebugEnabled() => boolean
begin
    return DBGEN == HIGH;
end;
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =====
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

func ExternalNoninvasiveDebugAllowed() => boolean
begin
    return ExternalNoninvasiveDebugAllowed\(PSTATE.EL\);
end;

// ExternalNoninvasiveDebugAllowed()
// =====

func ExternalNoninvasiveDebugAllowed(el : bits(2)) => boolean
begin
    if !ExternalNoninvasiveDebugEnabled\(\) then return FALSE; end;
    let ss : SecurityState = SecurityStateAtEL(el);

    if ((ELUsingAArch32(EL3) || ELUsingAArch32(EL1)) && el == EL0 &&
        ss == SS Secure && SDER().SUNIDEN == '1') then
        return TRUE;
    end;

    case ss of
        when SS NonSecure => return TRUE;
        when SS Secure => return ExternalSecureNoninvasiveDebugEnabled\(\);
        when SS Realm => return ExternalRealmNoninvasiveDebugEnabled\(\);
        when SS Root => return ExternalRootNoninvasiveDebugEnabled\(\);
    end;
end;
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =====
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.

readonly func ExternalNoninvasiveDebugEnabled() => boolean
begin
    return (IsFeatureImplemented(FEAT_Debugv8p4) || ExternalInvasiveDebugEnabled\(\) ||
        NIDEN == HIGH);
end;
```

Library pseudocode for shared/debug/authentication/ExternalRealmInvasiveDebugEnabled

```
// ExternalRealmInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

readonly func ExternalRealmInvasiveDebugEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_RME) then return FALSE; end;
    return ExternalInvasiveDebugEnabled\(\) && RLPIDEN == HIGH;
end;
```

Library pseudocode for shared/debug/authentication/ExternalRealmNoninvasiveDebugEnabled

```
// ExternalRealmNoninvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

readonly func ExternalRealmNoninvasiveDebugEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_RME) then return FALSE; end;
    return ExternalRealmInvasiveDebugEnabled\(\);
end;
```

Library pseudocode for shared/debug/authentication/ExternalRootInvasiveDebugEnabled

```
// ExternalRootInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND RTPIDEN AND SPIDEN) signal when FEAT_SEL2 is implemented
// and the (DBGEN AND RLPIDEN AND RTPIDEN) signal when FEAT_SEL2 is not implemented.

func ExternalRootInvasiveDebugEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_RME) then return FALSE; end;
    return (ExternalInvasiveDebugEnabled\(\) &&
        (!IsFeatureImplemented(FEAT_SEL2) || ExternalSecureInvasiveDebugEnabled\(\)) &&
        ExternalRealmInvasiveDebugEnabled\(\) &&
        RTPIDEN == HIGH);
end;
```

Library pseudocode for shared/debug/authentication/ExternalRootNoninvasiveDebugEnabled

```
// ExternalRootNoninvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.

func ExternalRootNoninvasiveDebugEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_RME) then return FALSE; end;
    return ExternalRootInvasiveDebugEnabled\(\);
end;
```

Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

readonly func ExternalSecureInvasiveDebugEnabled() => boolean
begin
    if !HaveSecureState\(\) then
        return IsFeatureImplemented(FEAT_RME) && RTPIDEN == HIGH;
    end;
    return ExternalInvasiveDebugEnabled\(\) && SPIDEN == HIGH;
end;
```

Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =====
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

readonly func ExternalSecureNoninvasiveDebugEnabled() => boolean
begin
    if !HaveSecureState\(\) then return FALSE; end;
    if !IsFeatureImplemented(FEAT_Debugv8p4) then
        return (ExternalNoninvasiveDebugEnabled\(\) &&
            (SPIDEN == HIGH || SPNIDEN == HIGH));
    else
        return ExternalSecureInvasiveDebugEnabled\(\);
    end;
end;
```

Library pseudocode for shared/debug/authentication/InvasiveDebugPermittedPAS

```
// InvasiveDebugPermittedPAS()
// =====
// Returns TRUE if the invasive debug of the configured PASpace is permitted by
// the authentication interface, and FALSE otherwise.

func InvasiveDebugPermittedPAS(pas : PASpace) => boolean
begin
  case pas of
    when PAS\_Secure => return ExternalSecureInvasiveDebugEnabled\(\);
    when PAS\_NonSecure => return ExternalInvasiveDebugEnabled\(\);
    when PAS\_Root => return ExternalRootInvasiveDebugEnabled\(\);
    when PAS\_Realm => return ExternalRealmInvasiveDebugEnabled\(\);
    otherwise => return FALSE;
  end;
end;
```

Library pseudocode for shared/debug/authentication/IsAccessNonSecure

```
// IsAccessNonSecure()
// =====
// Returns TRUE when an access is Non-Secure

func IsAccessNonSecure(addrdesc : AddressDescriptor) => boolean
begin
  return addrdesc.paddress.paspace == PAS\_NonSecure;
end;
```

Library pseudocode for shared/debug/authentication/IsAccessRealm

```
// IsAccessRealm()
// =====
// Returns TRUE when an access is Realm

func IsAccessRealm(addrdesc : AddressDescriptor) => boolean
begin
  return addrdesc.paddress.paspace == PAS\_Realm;
end;
```

Library pseudocode for shared/debug/authentication/IsAccessRoot

```
// IsAccessRoot()
// =====
// Returns TRUE when an access is Root

func IsAccessRoot(addrdesc : AddressDescriptor) => boolean
begin
  return addrdesc.paddress.paspace == PAS\_Root;
end;
```

Library pseudocode for shared/debug/authentication/IsAccessSecure

```
// IsAccessSecure()
// =====
// Returns TRUE when an access is Secure

func IsAccessSecure(addrdesc : AddressDescriptor) => boolean
begin
  return addrdesc.paddress.paspace == PAS\_Secure;
end;
```

Library pseudocode for shared/debug/authentication/IsCorePowered

```
// IsCorePowered()
// =====
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.

impdef func IsCorePowered() => boolean
begin
  return TRUE;
end;
```

Library pseudocode for shared/debug/authentication/IsMostSecureAccess

```
// IsMostSecureAccess()
// =====
// Returns TRUE if the security state of an access is the most secure PAS.

func IsMostSecureAccess(addrdesc : AddressDescriptor) => boolean
begin
    if IsFeatureImplemented(FEAT_RME) then
        return addrdesc.paddress.paspace == PAS\_Root;
    elsif HaveEL\(EL3\) || SecureOnlyImplementation\(\) then
        return addrdesc.paddress.paspace == PAS\_Secure;
    else
        assert addrdesc.paddress.paspace == PAS\_NonSecure;
        return TRUE;
    end;
end;
```

Library pseudocode for shared/debug/authentication/IsPASValid

```
// IsPASValid()
// =====
// Returns TRUE if the given value of 'pas' is not reserved, and FALSE otherwise.

func IsPASValid(pas : bits(2)) => boolean
begin
    case pas of
        when '00' => return IsFeatureImplemented(FEAT_Secure);
        when '01' => return TRUE;
        when '10' => return IsFeatureImplemented(FEAT_RME);
        when '11' => return IsFeatureImplemented(FEAT_RME);
    end;
end;
```

Library pseudocode for shared/debug/breakpoint/BreakpointInfo

```
// BreakpointInfo
// =====
// Breakpoint related fields.

type BreakpointInfo of record {
    bptype : BreakpointType, // Type of breakpoint matched
    match : boolean,         // breakpoint match
    mismatch : boolean       // breakpoint mismatch
};
```

Library pseudocode for shared/debug/breakpoint/BreakpointType

```
// BreakpointType
// =====

type BreakpointType of enumeration {
    BreakpointType_Inactive, // Breakpoint inactive or disabled
    BreakpointType_AddrMatch, // Address Match breakpoint
    BreakpointType_AddrMismatch, // Address Mismatch breakpoint
    BreakpointType_CtxtMatch }; // Context matching breakpoint
```

Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

```
// CheckValidStateMatch()
// =====
// Checks for an invalid state match that will generate Constrained
// Unpredictable behavior, otherwise returns Constraint_NONE.

func CheckValidStateMatch(ssc_in : bits(2), ssce_in : bit,
                          hmc_in : bit, pxc_in : bits(2),
                          isbreakpnt : boolean) => (Constraint, bits(2), bit, bit, bits(2))

begin
    if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0'; end;
    var reserved : boolean = FALSE;
    var ssc : bits(2) = ssc_in;
    var ssce : bit = ssce_in;
    var hmc : bit = hmc_in;
    var pxc : bits(2) = pxc_in;

    // Values that are not allocated in any architecture version
    case hmc::ssce::ssc::pxc of
        when '0 0 11 10' => reserved = TRUE;
        when '0 0 1x xx' => reserved = !HaveSecureState();
        when '1 0 00 x0' => reserved = TRUE;
        when '1 0 01 10' => reserved = TRUE;
        when '1 0 1x 10' => reserved = TRUE;
        when 'x 1 xx xx' => reserved = ssc != '01' || (hmc::pxc) IN {'000','110'};
        otherwise => reserved = FALSE;
    end;

    // Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && hmc::pxc == '000' && ssc != '11' then
        reserved = TRUE;
    end;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (hmc != '0' || ssc != '00') then
        reserved = TRUE;
    end;

    // EL3 is not implemented
    if !HaveEL(EL3) && ssc IN {'01','10'} && hmc::ssc::pxc != '10100' then
        reserved = TRUE;
    end;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && hmc::ssc::pxc == '11000' then
        reserved = TRUE;
    end;

    // EL2 is not implemented
    if !HaveEL(EL2) && hmc::ssc::pxc == '11100' then
        reserved = TRUE;
    end;

    // Secure EL2 is not implemented
    if !IsFeatureImplemented(FEAT_SEL2) && (hmc::ssc::pxc) IN {'01100','10100','x11x1'} then
        reserved = TRUE;
    end;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        var c : Constraint;
        var unpred_state_bits : bits(6);
        (c, unpred_state_bits) = ConstrainUnpredictableBits{6}(Unpredictable_RESBPWCTRL);
        hmc = unpred_state_bits[5];
        ssc = unpred_state_bits[4:3];
        ssce = unpred_state_bits[2];
        pxc = unpred_state_bits[1:0];
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, ARBITRARY : bits(2), ARBITRARY : bit, ARBITRARY : bit, ARBITRARY : bits(2));
        end;
    end;

    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, ssc, ssce, hmc, pxc);
end;
```

Library pseudocode for shared/debug/breakpoint/ContextAwareBreakpointRange

```
// ContextAwareBreakpointRange()
// =====
// Returns two numbers indicating the index of the first and last context-aware breakpoint.

func ContextAwareBreakpointRange() => (integer, integer)
begin
    let b : integer = NumBreakpointsImplemented();
    let c : integer = NumContextAwareBreakpointsImplemented();

    if b <= 16 then
        return (b - c, b - 1);
    elsif c <= 16 then
        return (16 - c, 15);
    else
        return (0, c - 1);
    end;
end;
```

Library pseudocode for shared/debug/breakpoint/IsContextAwareBreakpoint

```
// IsContextAwareBreakpoint()
// =====
// Returns TRUE if DBGBCR_EL1[n] is a context-aware breakpoint.

func IsContextAwareBreakpoint(n : integer) => boolean
begin
    let (lower, upper) : (integer, integer) = ContextAwareBreakpointRange();
    return n >= lower && n <= upper;
end;
```

Library pseudocode for shared/debug/breakpoint/NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// =====
// Returns the number of breakpoints implemented.

readonly func NumBreakpointsImplemented() => integer
begin
    return ImpDefInt("Number of breakpoints");
end;
```

Library pseudocode for shared/debug/breakpoint/NumContextAwareBreakpointsImplemented

```
// NumContextAwareBreakpointsImplemented()
// =====
// Returns the number of context-aware breakpoints implemented.

readonly func NumContextAwareBreakpointsImplemented() => integer
begin
    return ImpDefInt("Number of context-aware breakpoints");
end;
```

Library pseudocode for shared/debug/breakpoint/NumWatchpointsImplemented

```
// NumWatchpointsImplemented()
// =====
// Returns the number of watchpoints implemented.

readonly func NumWatchpointsImplemented() => integer
begin
    return ImpDefInt("Number of watchpoints");
end;
```

Library pseudocode for shared/debug/cti/CTI_ProcessEvent

```
// CTI_ProcessEvent()
// =====
// Process a discrete event on a Cross Trigger output event trigger.

impdef func CTI_ProcessEvent(id : CrossTriggerOut)
begin
    return;
end;
```

Library pseudocode for shared/debug/cti/CTI_SetEventLevel

```
// CTI_SetEventLevel()
// =====
// Set a Cross Trigger multi-cycle input event trigger to the specified level.

impdef func CTI_SetEventLevel(id : CrossTriggerIn, level : Signal)
begin
    return;
end;
```

Library pseudocode for shared/debug/cti/CTI_SignalEvent

```
// CTI_SignalEvent()
// =====
// Signal a discrete event on a Cross Trigger input event trigger.

impdef func CTI_SignalEvent(id : CrossTriggerIn)
begin
    return;
end;
```

Library pseudocode for shared/debug/cti/CrossTrigger

```
// CrossTrigger
// =====

type CrossTriggerOut of enumeration {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                                     CrossTriggerOut_IRQ,           CrossTriggerOut_RSVD3,
                                     CrossTriggerOut_TraceExtIn0,    CrossTriggerOut_TraceExtIn1,
                                     CrossTriggerOut_TraceExtIn2,    CrossTriggerOut_TraceExtIn3};

type CrossTriggerIn of enumeration {CrossTriggerIn_CrossHalt,      CrossTriggerIn_PMUOverflow,
                                    CrossTriggerIn_SPESample,      CrossTriggerIn_RSVD3,
                                    CrossTriggerIn_TraceExtOut0,     CrossTriggerIn_TraceExtOut1,
                                    CrossTriggerIn_TraceExtOut2,     CrossTriggerIn_TraceExtOut3,
                                    CrossTriggerIn_TRBESStop,        CrossTriggerIn_TRBEMgmt,
                                    CrossTriggerIn_TRBEWrap};
```

Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =====

func CheckForDCCInterrupts()
begin
    let commrx : boolean = (EDSCR().RXfull == '1');
    let commtx : boolean = (EDSCR().TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    var commirq : boolean;
    if ELUsingAArch32 (EL1) then
        commirq = ((commrx && DBGDCCINT().RX == '1') ||
                  (commtx && DBGDCCINT().TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1().RX == '1') ||
                  (commtx && MDCCINT_EL1().TX == '1'));
    end;
    SetInterruptRequestLevel(InterruptID\_COMMIRQ, if commirq then HIGH else LOW);

    return;
end;
```

Library pseudocode for shared/debug/dccanditr/DTR

```
// DTR
// ===

var DTRRX : bits(32);
var DTRTX : bits(32);
```


Library pseudocode for shared/debug/dccanditr/Read_DBGDTRRX_EL0

```
// Read_DBGDTRRX_EL0()
// =====
// Called on reads of debug register 0x080.

func Read_DBGDTRRX_EL0(memory_mapped : boolean) => bits(32)
begin
    return DTRRX;
end;
```

Library pseudocode for shared/debug/dccanditr/Read_DBGDTRTX_EL0

```
// Read_DBGDTRTX_EL0()
// =====
// Called on reads of debug register 0x08C.

func Read_DBGDTRTX_EL0(memory_mapped : boolean) => bits(32)
begin
    let underrun : boolean = (EDSCR().TXfull == '0' ||
                              (Halted() && EDSCR().MA == '1' && EDSCR().ITE == '0'));
    let value : bits(32) = if underrun then ARBITRARY : bits(32) else DTRTX;

    if EDSCR().ERR == '1' then return value; end;          // Error flag set: no side-effects

    if underrun then
        EDSCR().TXU = '1'; EDSCR().ERR = '1';             // Underrun condition: block side-effects
        return value;                                     // Return UNKNOWN
    end;

    EDSCR().TXfull = '0';
    if Halted() && EDSCR().MA == '1' then
        EDSCR().ITE = '0';                                // See comments in Write_EDITR()

        if !UsingAArch32() then
            ExecuteA64(0xB8404401[31:0]);                  // A64 "LDR W1, [X0], #4"
        else
            ExecuteT32(0xF850[15:0] /*hw1*/, 0x1B04[15:0] /*hw2*/); // T32 "LDR R1, [R0], #4"
        end;
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR().ERR == '1' then
            EDSCR().TXfull = ARBITRARY : bit;
            DBGDTRTX_EL0() = ARBITRARY : bits(64);
        else

            if !UsingAArch32() then
                ExecuteA64(0xD5130501[31:0]);              // A64 "MSR_DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00[15:0] /*hw1*/, 0x1E15[15:0] /*hw2*/); // T32 "MSR_DBGDTRTXint,R1"
            end;
            // "MSR_DBGDTRTX_EL0,X1" calls Write_DBGDTR_EL0() which sets TXfull.
            assert EDSCR().TXfull == '1';

        end;
        if !UsingAArch32() then
            X{64}(1) = ARBITRARY : bits(64);
        else
            R(1) = ARBITRARY : bits(32);
        end;
        EDSCR().ITE = '1';                                // See comments in Write_EDITR()
    end;

    return value;
end;
```

Library pseudocode for shared/debug/dccanditr/Read_DBGDTR_EL0

```
// Read_DBGDTR_EL0()
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

func Read_DBGDTR_EL0{N}() => bits(N)
begin
    // For MRS <Rt>,DBGDTRTX_EL0 N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0 N=64, X[t]=result
    assert N IN {32,64};
    var result : bits(N);
    if EDSCR().RXfull == '0' then
        result = ARBITRARY : bits(N);
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result[63:32] = DTRTX; end;
        result[31:0] = DTRRX;
    end;
    EDSCR().RXfull = '0';
    return result;
end;
```

Library pseudocode for shared/debug/dccanditr/Write_DBGDTRRX_EL0

```
// Write_DBGDTRRX_EL0()
// =====
// Called on writes to debug register 0x080.

func Write_DBGDTRRX_EL0(memory_mapped : boolean, value : bits(32))
begin
    if EDSR().ERR == '1' then return; end; // Error flag set: ignore write

    if EDSR().RXfull == '1' || (Halted() && EDSR().MA == '1' && EDSR().ITE == '0') then
        EDSR().RXO = '1'; EDSR().ERR = '1'; // Overrun condition: ignore write
        return;
    end;

    EDSR().RXfull = '1';
    DTRRX = value;

    if Halted() && EDSR().MA == '1' then
        EDSR().ITE = '0'; // See comments in Write_EDITR()
        if !UsingAArch32() then
            ExecuteA64(0xD5330501[31:0]); // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401[31:0]); // A64 "STR W1,[X0],#4"
            X{64}(1) = ARBITRARY : bits(64);
        else
            ExecuteT32(0xEE10[15:0] /*hw1*/, 0x1E15[15:0] /*hw2*/); // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840[15:0] /*hw1*/, 0x1B04[15:0] /*hw2*/); // T32 "STR R1,[R0],#4"
            R(1) = ARBITRARY : bits(32);
        end;
        // If the store aborts, the Data Abort exception is taken and EDSR.ERR is set to 1
        if EDSR().ERR == '1' then
            EDSR().RXfull = ARBITRARY : bit;
            DBGDTRRX_EL0() = ARBITRARY : bits(64);
        else
            // "MRS X1,DBGDTRRX_EL0" calls Read_DBGDTR_EL0() which clears RXfull.
            assert EDSR().RXfull == '0';
        end;

        EDSR().ITE = '1'; // See comments in Write_EDITR()
    end;
    return;
end;
```

Library pseudocode for shared/debug/dccanditr/Write_DBGDTRTX_EL0

```
// Write_DBGDTRTX_EL0()
// =====
// Called on writes to debug register 0x08C.

func Write_DBGDTRTX_EL0(memory_mapped : boolean, value : bits(32))
begin
    DTRTX = value;
    return;
end;
```

Library pseudocode for shared/debug/dccanditr/Write_DBGDTR_EL0

```
// Write_DBGDTR_EL0()
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

func Write_DBGDTR_EL0{N}(value_in : bits(N))
begin
    var value : bits(N) = value_in;
    // For MSR DBGDTRTX_EL0,<Rt> N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt> N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSR().TXfull == '1' then
        value = ARBITRARY : bits(N);
    end;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value[63:32]; end;
    DTRTX = value[31:0]; // 32-bit or 64-bit write
    EDSR().TXfull = '1';
    return;
end;
```

Library pseudocode for shared/debug/dccanditr/Write_EDITR

```
// Write_EDITR()
// =====
// Called on writes to debug register 0x084.

func Write_EDITR(memory_mapped : boolean, value : bits(32))
begin
    if EDSCR().ERR == '1' then return; end;           // Error flag set: ignore write

    if !Halted() then return; end;                   // Non-debug state: ignore write

    if EDSCR().ITE == '0' || EDSCR().MA == '1' then
        EDSCR().ITO = '1'; EDSCR().ERR = '1';        // Overrun condition: block write
        return;
    end;

    // ITE indicates whether the PE is ready to accept another instruction; the PE
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR().ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value[15:0]/*hw1*/, value[31:16] /*hw2*/);
    end;

    EDSCR().ITE = '1';

    return;
end;
```



```

// DCPSInstruction()
// =====
// Operation of the DCPS instruction in Debug state

func DCPSInstruction(target_el : bits(2))
begin

    SynchronizeContext();

    var handle_el : bits(2);
    case target_el of
        when EL1 =>
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
                handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2().TGE == '1' then
                Undefined();
            else
                handle_el = EL1;
            end;
        when EL2 =>
            if !HaveEL(EL2) then
                Undefined();
            elsif PSTATE.EL == EL3 && !UsingAArch32() then
                handle_el = EL3;
            elsif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
                Undefined();
            else
                handle_el = EL2;
            end;
        when EL3 =>
            if EDSCR().SDD == '1' || !HaveEL(EL3) then
                Undefined();
            else
                handle_el = EL3;
            end;
        otherwise =>
            unreachable;
    end;

    let from_secure : boolean = CurrentSecurityState() == SS_Secure;
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR().NS = '0'; end;
        assert UsingAArch32(); // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1 =>
                AArch32_WriteMode(M32_Svc);
                if IsFeatureImplemented(FEAT_PAN) && SCTLR().SPAN == '0' then
                    PSTATE.PAN = '1';
                end;
            when EL2 => AArch32_WriteMode(M32_Hyp);
            when EL3 => AArch32_WriteMode(M32_Monitor);
                if IsFeatureImplemented(FEAT_PAN) then
                    if !from_secure then
                        PSTATE.PAN = '0';
                    elsif SCTLR().SPAN == '0' then
                        PSTATE.PAN = '1';
                    end;
                end;
        end;
        if handle_el == EL2 then
            ELR_hyp() = ARBITRARY : bits(32); HSR() = ARBITRARY : bits(32);
        else
            LR() = ARBITRARY : bits(32);
        end;
        SPSR_curr() = ARBITRARY : bits(32);
        if handle_el == EL2 then
            PSTATE.E = HSCTLR().EE;
        else
            PSTATE.E = SCTLR().EE;
        end;
        DLR() = ARBITRARY : bits(32); DSPSR() = ARBITRARY : bits(32);
    else // Targeting AArch64
        let from_32 : boolean = UsingAArch32();
        if from_32 then AArch64_MaybeZeroRegisterUppers(); end;
        if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
            ResetSVEState();
        else
            MaybeZeroSVEUppers(target_el);
        end;
        PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
        if IsFeatureImplemented(FEAT_PAN) && ((handle_el == EL1 && SCTLR_EL1().SPAN == '0') ||
            (handle_el == EL2 && ELIsInHost(EL0) &&
                SCTLR_EL2().SPAN == '0')) then
            PSTATE.PAN = '1';
        end;
        ELR_ELx() = ARBITRARY : bits(64);
        SPSR_ELx() = ARBITRARY : bits(64);
    end;
end

```

```

    ESR_ELx() = ARBITRARY : bits(64);
    DLR_EL0() = ARBITRARY : bits(64); DSPSR_EL0() = ARBITRARY : bits(64);
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0'; end;
    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1'; end;
    if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = '0'; end;
end;
if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
UpdateEDSCRFields(); // Update EDSCR PE state flags
return;
end;

```

Library pseudocode for shared/debug/halting/DRPSInstruction

```

// DRPSInstruction()
// =====
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

func DRPSInstruction()
begin
    SynchronizeContext();

    DebugRestorePSR();

    return;
end;

```

Library pseudocode for shared/debug/halting/DebugHalt

```

// DebugHalt
// =====
// Reason codes for entry to Debug state

constant DebugHalt_Breakpoint : bits(6)      = '000111';
constant DebugHalt_EDBGRQ : bits(6)          = '010011';
constant DebugHalt_Step_Normal : bits(6)      = '011011';
constant DebugHalt_Step_Exclusive : bits(6)   = '011111';
constant DebugHalt_OSUnlockCatch : bits(6)    = '100011';
constant DebugHalt_ResetCatch : bits(6)       = '100111';
constant DebugHalt_Watchpoint : bits(6)       = '101011';
constant DebugHalt_HaltInstruction : bits(6)   = '101111';
constant DebugHalt_SoftwareAccess : bits(6)   = '110011';
constant DebugHalt_ExceptionCatch : bits(6)   = '110111';
constant DebugHalt_Step_NoSyndrome : bits(6)  = '111011';

```

Library pseudocode for shared/debug/halting/DebugRestorePSR

```

// DebugRestorePSR()
// =====

func DebugRestorePSR()
begin
    // PSTATE.[N,Z,C,V,Q,GE,SS,D,A,I,F] are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        let spsr : bits(32) = SPSR_curr();
        SetPSTATEFromPSR{32}(spsr);
        PSTATE.[N,Z,C,V,Q,GE,SS,A,I,F] = ARBITRARY : bits(13);
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
        DLR() = ARBITRARY : bits(32); DSPSR() = ARBITRARY : bits(32);
    else
        let spsr : bits(64) = SPSR_ELx();
        SetPSTATEFromPSR{64}(spsr);
        PSTATE.[N,Z,C,V,SS,D,A,I,F] = ARBITRARY : bits(9);
        DLR_EL0() = ARBITRARY : bits(64); DSPSR_EL0() = ARBITRARY : bits(64);
    end;
    UpdateEDSCRFields(); // Update EDSCR PE state flags
end;

```

Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```

// DisableITRAndResumeInstructionPrefetch()
// =====

impdef func DisableITRAndResumeInstructionPrefetch()
begin
    return;
end;

```

Library pseudocode for shared/debug/halting/ExecuteA64

```
// ExecuteA64()  
// =====  
// Execute an A64 instruction in Debug state.  
  
impdef func ExecuteA64(instr : bits(32))  
begin  
    Unimplemented();  
end;
```

Library pseudocode for shared/debug/halting/ExecuteT32

```
// ExecuteT32()  
// =====  
// Execute a T32 instruction in Debug state.  
  
impdef func ExecuteT32(hw1 : bits(16), hw2 : bits(16))  
begin  
    Unimplemented();  
end;
```

Library pseudocode for shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// =====

func ExitDebugState()
begin
    assert Halted\(\);
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR().STATUS = '000001'; // Signal restarting
    // Clear any pending Halting debug events
    if IsFeatureImplemented(FEAT_Debugv8p8) then
        EDESR()[3:0] = '0000';
    else
        EDESR()[2:0] = '000';
    end;

    var new_pc : bits(64);
    var spsr : bits(64);

    if UsingAArch32\(\) then
        new_pc = ZeroExtend{64}(DLR());
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            spsr = DSPSR2() :: DSPSR();
        else
            spsr = ZeroExtend{64}(DSPSR());
        end;
    else
        new_pc = DLR_EL0();
        spsr = DSPSR_EL0();
    end;

    let illegal_psr_state : boolean = IllegalExceptionReturn{64}(spsr);
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR{64}(spsr); // Can update privileged bits, even at EL0

    let branch_conditional : boolean = FALSE;
    if UsingAArch32\(\) then
        if ConstrainUnpredictableBool(Unpredictable\_RESTARTALIGNPC) then new_pc[0] = '0'; end;
        // AArch32 branch
        BranchTo{32}(new_pc[31:0], BranchType\_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN.
        if illegal_psr_state && spsr[4] == '1' then
            new_pc[63:32] = ARBITRARY : bits(32);
            new_pc[1:0] = ARBITRARY : bits(2);
        end;
        if IsFeatureImplemented(FEAT_BRBE) then
            BRBEDebugStateExit(AArch64\_BranchAddr(new_pc, PSTATE.EL));
        end;
        // A type of branch that is never predicted
        BranchTo{64}(new_pc, BranchType\_DBGEXIT, branch_conditional);
    end;

    // Atomically signal restarted
    EDSCR().STATUS = '000010';
    EDPRSR().SDR = '1';
    // End of atomically signal
    EDPRSR().HALTED = '0';
    UpdateEDSCRFields(); // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
end;
```



```

// Halt()
// =====

func Halt(reason : bits(6))
begin
    let is_async : boolean = FALSE;
    let fault : FaultRecord = NoFault();
    Halt(reason, is_async, fault);
end;

// Halt()
// =====

func Halt(reason : bits(6), is_async : boolean, fault : FaultRecord)
begin
    CTI_SignalEvent(CrossTriggerIn\_CrossHalt); // Trigger other cores to halt

    let preferred_restart_address : bits(64) = ThisInstrAddr{}();
    var spsr : bits(64) = GetPSRFromPSTATE{64}(DebugState);

    if (IsFeatureImplemented(FEAT_BTI) && !is_async &&
        ! reason IN {DebugHalt\_Step\_Normal, DebugHalt\_Step\_Exclusive,
                     DebugHalt\_Step\_NoSyndrome, DebugHalt\_Breakpoint,
                     DebugHalt\_HaltInstruction} &&
        ConstrainUnpredictableBool(Unpredictable\_ZEROBTYPE)) then
        spsr[11:10] = '00';
    end;

    if UsingAArch32() then
        DLR() = preferred_restart_address[31:0];
        DSPSR() = spsr[31:0];
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DSPSR2() = spsr[63:32];
        end;
    else
        DLR_EL0() = preferred_restart_address;
        DSPSR_EL0() = spsr;
    end;
    EDSCR().ITE = '1';
    EDSCR().ITO = '0';
    if IsFeatureImplemented(FEAT_RME) then
        if PSTATE.EL == EL3 then
            EDSCR().SDD = '0';
        else
            EDSCR().SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
        end;
    elseif CurrentSecurityState() == SS\_Secure then
        EDSCR().SDD = '0'; // If entered in Secure state, allow debug
    elseif HaveEL(EL3) then
        EDSCR().SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        EDSCR().SDD = '1'; // Otherwise EDSCR.SDD is RES1
    end;
    EDSCR().MA = '0';

    // In Debug state:
    // * PSTATE.[SS,SSBS,D,A,I,F] are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.[N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT] are also not observable, but since these
    //   are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.[IT,T] are ignored.
    // * PSTATE.IL is ignored and behave-as-if 0.
    // * PSTATE.BTYPE is ignored and behave-as-if 0.
    // * PSTATE.TCO is set 1.
    // * PSTATE.PACM is ignored and behave-as-if 0.
    // * PSTATE.[UAO,PAN] are observable and not changed on entry into Debug state.
    // * PSTATE.UINJ is set to 0.

    if UsingAArch32() then
        PSTATE.[IT,SS,SSBS,A,I,F,T] = ARBITRARY : bits(14);
    else
        PSTATE.[SS,SSBS,D,A,I,F] = ARBITRARY : bits(6);
    end;

    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1'; end;
    if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00'; end;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0'; end;
    PSTATE.IL = '0';
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
    if IsFeatureImplemented(FEAT_BRBE) then
        BRBEDebugStateEntry(preferred_restart_address);
    end;
    StopInstructionPrefetchAndEnableITR();
    // atomic write
    EDSCR().STATUS = reason;
    EDPSR().HALTED = '1';
    // end of atomic write
    UpdateEDSCRFields(); // Update EDSCR PE state flags.
    if IsFeatureImplemented(FEAT_EDHCSR) then
        UpdateEDHCSR(reason, fault); // Update EDHCSR fields.
    end;
end;

```

```

end;
if !is_async then EndOfInstruction(EndOfInstructionReason Exception); end;
return;
end;

```

Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```

// HaltOnBreakpointOrWatchpoint()
// =====
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

func HaltOnBreakpointOrWatchpoint() => boolean
begin
    return HaltingAllowed() && EDSCR().HDE == '1' && OSLSR_EL1().OSLK == '0';
end;

```

Library pseudocode for shared/debug/halting/Halted

```

// Halted()
// =====

readonly func Halted() => boolean
begin
    return ! EDSCR().STATUS IN {'000001', '000010'};           // Halted
end;

```

Library pseudocode for shared/debug/halting/HaltingAllowed

```

// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

func HaltingAllowed() => boolean
begin
    if Halted() || DoubleLockStatus() then
        return FALSE;
    end;
    let ss : SecurityState = CurrentSecurityState();
    case ss of
        when SS\_NonSecure => return ExternalInvasiveDebugEnabled();
        when SS\_Secure =>    return ExternalSecureInvasiveDebugEnabled();
        when SS\_Root =>      return ExternalRootInvasiveDebugEnabled();
        when SS\_Realm =>     return ExternalRealmInvasiveDebugEnabled();
    end;
end;

```

Library pseudocode for shared/debug/halting/Restarting

```

// Restarting()
// =====

readonly func Restarting() => boolean
begin
    return EDSCR().STATUS == '000001';           // Restarting
end;

```

Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```

// StopInstructionPrefetchAndEnableITR()
// =====

impdef func StopInstructionPrefetchAndEnableITR()
begin
    return;
end;

```

Library pseudocode for shared/debug/halting/UpdateDbgAuthStatus

```
// UpdateDbgAuthStatus()
// =====
// Provides information about the state of the
// IMPLEMENTATION DEFINED authentication interface for debug.

func UpdateDbgAuthStatus()
begin
    var nsid, nsnid : bits(2);
    var sid, snid : bits(2);
    var rlid, rtid : bits(2);
    if SecureOnlyImplementation\(\) then
        nsid = '00';
    elseif ExternalInvasiveDebugEnabled\(\) then
        nsid = '11';          // Non-secure Invasive debug implemented and enabled.
    else
        nsid = '10';          // Non-secure Invasive debug implemented and disabled.
    end;

    if SecureOnlyImplementation\(\) then
        nsnid = '00';
    elseif ExternalNoninvasiveDebugEnabled\(\) then
        nsnid = '11';         // Non-secure Non-Invasive debug implemented and enabled.
    else
        nsnid = '10';         // Non-secure Non-Invasive debug implemented and disabled.
    end;

    if !HaveSecureState\(\) then
        sid = '00';
    elseif ExternalSecureInvasiveDebugEnabled\(\) then
        sid = '11';           // Secure Invasive debug implemented and enabled.
    else
        sid = '10';           // Secure Invasive debug implemented and disabled.
    end;

    if !HaveSecureState\(\) then
        snid = '00';
    elseif ExternalSecureNoninvasiveDebugEnabled\(\) then
        snid = '11';          // Secure Non-Invasive debug implemented and enabled.
    else
        snid = '10';          // Secure Non-Invasive debug implemented and disabled.
    end;

    if !IsFeatureImplemented\(FEAT\_RME\) then
        rlid = '00';
    elseif ExternalRealmInvasiveDebugEnabled\(\) then
        rlid = '11';          // Realm Invasive debug implemented and enabled.
    else
        rlid = '10';          // Realm Invasive debug implemented and disabled.
    end;

    if !IsFeatureImplemented\(FEAT\_RME\) then
        rtid = '00';
    elseif ExternalRootInvasiveDebugEnabled\(\) then
        rtid = '11';          // Root Invasive debug implemented and enabled.
    else
        rtid = '10';          // Root Invasive debug implemented and disabled.
    end;

    DBGAUTHSTATUS_EL1().NSID = nsid;
    DBGAUTHSTATUS_EL1().NSNID = nsnid;
    DBGAUTHSTATUS_EL1().SID = sid;
    DBGAUTHSTATUS_EL1().SNID = snid;
    DBGAUTHSTATUS_EL1().RLID = rlid;
    DBGAUTHSTATUS_EL1().RLNID = rlid;          // Field has the same value as DBGAUTHSTATUS_EL1.RLID.
    DBGAUTHSTATUS_EL1().RTID = rtid;
    DBGAUTHSTATUS_EL1().RTNID = rtid;          // Field has the same value as DBGAUTHSTATUS_EL1.RTID.
    return;
end;
```

Library pseudocode for shared/debug/halting/UpdateEDHSR

```
// UpdateEDHSR()
// =====
// Update EDHSR watchpoint related fields.

func UpdateEDHSR(reason : bits(6), fault : FaultRecord)
begin
    var syndrome : bits(64) = Zeros{};
    if reason == DebugHalt Watchpoint then
        if IsFeatureImplemented(FEAT_GCS) && fault.accessdesc.acctype == AccessType\_GCS then
            syndrome[40] = '1'; // GCS
        end;
        syndrome[23:0] = WatchpointRelatedSyndrome(fault);
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            if fault.write then syndrome[6] = '1'; end; // WnR
            if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
                syndrome[8] = '1'; // CM
            end;
            if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType\_NV2 then
                syndrome[13] = '1'; // VNCR
            end;
        end;
    else
        syndrome = ARBITRARY : bits(64);
    end;

    EDHSR() = syndrome;
end;
```

Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// =====
// Update EDSCR PE state fields

func UpdateEDSCRFields()
begin
    if !Halted() then
        EDSCR().EL = '00';
        if IsFeatureImplemented(FEAT_RME) then
            // SDD bit.
            EDSCR().SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
            EDSCR().[NSE,NS] = ARBITRARY : bits(2);
        else
            // SDD bit.
            EDSCR().SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
            EDSCR().NS = ARBITRARY : bit;
        end;

        EDSCR().RW = '1111';
    else
        EDSCR().EL = PSTATE.EL;
        // SError Pending.
        if EL2Enabled() && HCR_EL2().[AMO,TGE] == '10' && PSTATE.EL IN {EL0,EL1} then
            EDSCR().A = if IsVirtualSErrorPending() then '1' else '0';
        elsif (EffectiveSCR_EL3.EnDSE() == '1' && PSTATE.EL IN {EL0,EL1,EL2}) then
            EDSCR().A = if IsDelegatedSErrorPending() then '1' else '0';
        else
            EDSCR().A = if IsPhysicalSErrorPending() then '1' else '0';
        end;

        let ss : SecurityState = CurrentSecurityState();
        if IsFeatureImplemented(FEAT_RME) then
            case ss of
                when SS_Secure => EDSCR().[NSE,NS] = '00';
                when SS_NonSecure => EDSCR().[NSE,NS] = '01';
                when SS_Root => EDSCR().[NSE,NS] = '10';
                when SS_Realm => EDSCR().[NSE,NS] = '11';
            end;
        else
            EDSCR().NS = if ss == SS_Secure then '0' else '1';
        end;

        var RW : bits(4);
        RW[1] = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW[0] = RW[1];
        else
            RW[0] = if UsingAArch32() then '0' else '1';
        end;
        if !EL2Enabled() then
            RW[2] = RW[1];
        else
            RW[2] = if ELUsingAArch32(EL2) then '0' else '1';
        end;
        if !HaveEL(EL3) then
            RW[3] = RW[2];
        else
            RW[3] = if ELUsingAArch32(EL3) then '0' else '1';
        end;

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW[3] == '0' then RW[2:0] = ARBITRARY : bits(3);
        elsif RW[2] == '0' then RW[1:0] = ARBITRARY : bits(2);
        elsif RW[1] == '0' then RW[0] = ARBITRARY : bit; end;
        EDSCR().RW = RW;
    end;
    return;
end;
```

Library pseudocode for shared/debug/halting/events

```
var EDBGREQ : Signal; // Input Signal in external debug interface
```

Library pseudocode for shared/debug/haltingevents/CheckEDBGRQ

```
// CheckEDBGRQ()
// =====
// Checks IMPLEMENTATION DEFINED EDBGRQ Input Signal of external debug interface.
// This is an example of an IMPLEMENTATION DEFINED source of External Debug Request debug events.

func CheckEDBGRQ()
begin
    if EDBGRQ == HIGH then
        ExternalDebugRequest();
    end;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current Exception level

func CheckExceptionCatch(exception_entry : boolean)
begin
    // Called after an exception entry or exit, that is, such that the Security state
    // and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
    // is not implemented, this function might also be called at any time.
    let ss : SecurityState = SecurityStateAtEL(PSTATE.EL);
    var base : integer;

    case ss of
        when SS Secure => base = 0;
        when SS NonSecure => base = 4;
        when SS Realm => base = 16;
        when SS Root => base = 0;
    end;
    if HaltingAllowed() then
        var halt : boolean;
        if IsFeatureImplemented(FEAT_Debugv8p2) then
            let exception_exit : boolean = !exception_entry;
            let increment : integer{} = if ss == SS Realm then 4 else 8;
            let ctrl : bits(2) = (EDECCECR()[UInt(PSTATE.EL) + base + increment]::
                EDECCECR()[UInt(PSTATE.EL) + base]);

            case ctrl of
                when '00' => halt = FALSE;
                when '01' => halt = TRUE;
                when '10' => halt = (exception_exit == TRUE);
                when '11' => halt = (exception_entry == TRUE);
            end;
        else
            halt = (EDECCECR()[UInt(PSTATE.EL) + base] == '1');
        end;

        if halt then
            if IsFeatureImplemented(FEAT_Debugv8p8) && exception_entry then
                EDESR().EC = '1';
            else
                Halt(DebugHalt\_ExceptionCatch);
            end;
        end;
    end;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckExternalDebugRequestEvents

```
// CheckExternalDebugRequestEvents()
// =====
// Checks for all External Debug Request debug events.

func CheckExternalDebugRequestEvents()
begin
    CheckTRBEHalt();
    CheckPMUHalt();
    CheckEDBGRQ();
    return;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step

func CheckHaltingStep(is_async : boolean)
begin
  let step_enabled : boolean = EDECR().SS == '1' && HaltingAllowed\(\);
  let active_pending : boolean = step_enabled && EDESR().SS == '1';
  if active_pending then
    if HaltingStep\_DidNotStep\(\) then
      let fault : FaultRecord = NoFault\(\);
      Halt\(DebugHalt\_Step\_NoSyndrome, is\_async, fault\);
    elseif HaltingStep\_SteppedEX\(\) then
      let fault : FaultRecord = NoFault\(\);
      Halt\(DebugHalt\_Step\_Exclusive, is\_async, fault\);
    else
      let fault : FaultRecord = NoFault\(\);
      Halt\(DebugHalt\_Step\_Normal, is\_async, fault\);
    end;
  end;
  if step_enabled then ShouldAdvanceHS = TRUE; end;
  return;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

func CheckOSUnlockCatch()
begin
  if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL().OSUCE == '1') ||
    (!IsFeatureImplemented(FEAT_DoPD) && EDECR().OSUCE == '1')) then
    if !Halted\(\) then EDESR().OSUC = '1'; end;
  end;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckPMUHalt

```
// CheckPMUHalt()
// =====

func CheckPMUHalt()
begin
  if !IsFeatureImplemented(FEAT_Debugv8p9) || !IsFeatureImplemented(FEAT_PMUv3p9) then
    return;
  end;

  // The request remains set until the condition is cleared.
  // For example, an interrupt handler or cross-triggered event handler clears
  // the overflow status flag by writing to PMOVSLR_EL0

  let include_r1 : boolean = TRUE;
  let include_r2 : boolean = TRUE;
  let include_r3 : boolean = TRUE;

  let pmuhalt : boolean = CheckPMUOverflowCondition(PMUOverflowCondition\_EDBGRQ,
    include_r1, include_r2, include_r3);

  if pmuhalt && EDECR().PME == '1' then
    ExternalDebugRequest\(\);
  end;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckPendingExceptionCatch

```
// CheckPendingExceptionCatch()
// =====
// Check whether EDESR.EC has been set by an Exception Catch debug event.

func CheckPendingExceptionCatch(is_async : boolean)
begin
  if IsFeatureImplemented(FEAT_Debugv8p8) && HaltingAllowed\(\) && EDESR().EC == '1' then
    let fault : FaultRecord = NoFault\(\);
    Halt\(DebugHalt\_ExceptionCatch, is\_async, fault\);
  end;
end;
```


Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// =====
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

func CheckPendingOSUnlockCatch()
begin
    if HaltingAllowed() && EDESR().OSUC == '1' then
        let is_async : boolean = TRUE;
        let fault : FaultRecord = NoFault();
        Halt(DebugHalt\_OSUnlockCatch, is_async, fault);
    end;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// =====
// Check whether EDESR.RC has been set by a Reset Catch debug event

func CheckPendingResetCatch()
begin
    if HaltingAllowed() && EDESR().RC == '1' then
        let is_async : boolean = TRUE;
        let fault : FaultRecord = NoFault();
        Halt(DebugHalt\_ResetCatch, is_async, fault);
    end;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =====
// Called after reset

func CheckResetCatch()
begin
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL().RCE == '1') ||
        (!IsFeatureImplemented(FEAT_DoPD) && EDECR().RCE == '1')) then
        EDESR().RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt\_ResetCatch); end;
    end;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====
// Check for access to Breakpoint and Watchpoint registers.

func CheckSoftwareAccessToDebugRegisters()
begin
    let os_lock : bit = (if ELUsingAArch32(EL1) then DBGOSLSR().OSLK else OSLSR_EL1().OSLK);
    if HaltingAllowed() && EDSCR().TDA == '1' && os_lock == '0' then
        Halt(DebugHalt\_SoftwareAccess);
    end;
end;
```

Library pseudocode for shared/debug/haltingevents/CheckTRBEHalt

```
// CheckTRBEHalt()
// =====

func CheckTRBEHalt()
begin
    if !IsFeatureImplemented(FEAT_Debugv8p9) || !IsFeatureImplemented(FEAT_TRBE_EXT) then
        return;
    end;

    if TraceBufferEnabled() && TRBSR_EL1().IRQ == '1' && EDECR().TRBE == '1' then
        ExternalDebugRequest();
    end;
end;
```

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// =====

func ExternalDebugRequest()
begin
    if HaltingAllowed() then
        let is_async : boolean = TRUE;
        let fault : FaultRecord = NoFault();
        Halt(DebugHalt\_EDBGRQ, is_async, fault);
        // Otherwise the CTI continues to assert the debug request until it is taken.
    end;
end;
```

Library pseudocode for shared/debug/haltingevents/HSAdvance

```
// HSAdvance()
// =====
// Advance the Halting Step State Machine

func HSAdvance()
begin
    if !ShouldAdvanceHS then return; end;
    let step_enabled : boolean = EDECR().SS == '1' && HaltingAllowed();
    let active_not_pending : boolean = step_enabled && EDESR().SS == '0';
    if active_not_pending then EDESR().SS = '1'; end; // set as pending.
    ShouldAdvanceHS = FALSE;
    return;
end;
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_DidNotStep

```
// HaltingStep_DidNotStep()
// =====
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.

impdef func HaltingStep_DidNotStep() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_SteppedEX

```
// HaltingStep_SteppedEX()
// =====
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.

impdef func HaltingStep_SteppedEX() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =====
// Determine whether EDSCR disables interrupts routed to 'target'.

func ExternalDebugInterruptsDisabled(target : bits(2)) => boolean
begin
    var int_dis : boolean;
    let ss : SecurityState = SecurityStateAtEL(target);
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if EDSCR().INTdis[0] == '1' then
            case ss of
                when SS_NonSecure => int_dis = ExternalInvasiveDebugEnabled();
                when SS_Secure => int_dis = ExternalSecureInvasiveDebugEnabled();
                when SS_Realm => int_dis = ExternalRealmInvasiveDebugEnabled();
                when SS_Root => int_dis = ExternalRootInvasiveDebugEnabled();
            end;
        else
            int_dis = FALSE;
        end;
    else
        case target of
            when EL3 =>
                int_dis = (EDSCR().INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
            when EL2 =>
                int_dis = (EDSCR().INTdis == '1x' && ExternalInvasiveDebugEnabled());
            when EL1 =>
                if ss == SS_Secure then
                    int_dis = (EDSCR().INTdis == '1x' && ExternalSecureInvasiveDebugEnabled());
                else
                    int_dis = (EDSCR().INTdis != '00' && ExternalInvasiveDebugEnabled());
                end;
            end;
        end;
    end;
    return int_dis;
end;
```

Library pseudocode for shared/debug/pmu

```
var PMUEventAccumulator : array [[31]] of integer; // Accumulates PMU events for a cycle
var PMULastThresholdValue : array [[31]] of boolean; // A record of the threshold result for each
// event counter on the previous cycle

// Constant used in PMU functions to represent actions on the cycle counter.
constant CYCLE_COUNTER_ID : integer = 31;
// Constant used in PMU functions to represent actions on the instruction counter.
constant INSTRUCTION_COUNTER_ID : integer = 32;
```

Library pseudocode for shared/debug/pmu/CheckForPMUOverflow

```
// CheckForPMUOverflow()
// =====
// Called before each instruction is executed.
// If a PMU event counter has overflowed, this function might do any of:
// - Signal a Performance Monitors overflow interrupt request.
// - Signal a CTI Performance Monitors overflow event.
// - Generate an External Debug Request debug event.
// - Generate a BRBE freeze event.

func CheckForPMUOverflow()
begin
    let include_r1 : boolean = TRUE;
    let include_r2 : boolean = TRUE;
    let include_r3 : boolean = TRUE;

    let enabled : boolean = PMUIInterruptEnabled();
    let pmuirq : boolean = CheckPMUOverflowCondition(PMUOverflowCondition_IRQ,
                                                    include_r1, include_r2, include_r3);

    SetInterruptRequestLevel(InterruptID_PMIIRQ,
                            if enabled && pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow,
                     if pmuirq then HIGH else LOW);

    if ShouldBRBEFreeze() then
        BRBEFreeze();
    end;

    return;
end;
```



```

// CheckPMUOverflowCondition()
// =====
// Checks for PMU overflow under certain parameter conditions described by 'reason'.
// If 'include_r1' is TRUE, then check counters in the range [0..(HPMN-1)], CCNTR
// and ICNTR, unless excluded by 'reason'.
// If 'include_r2' is TRUE, then check counters in the range [HPMN..(EPMN-1)].
// If 'include_r3' is TRUE, then check counters in the range [EPMN..(N-1)].

func CheckPMUOverflowCondition(reason : PMUOverflowCondition,
                               include_r1 : boolean, include_r2 : boolean, include_r3 : boolean
                               ) => boolean
begin
    // 'reason' is decoded into a further set of parameters:
    // If 'check_e' is TRUE, then check the applicable one of PMCR_EL0.E and MDCR_EL2.HPME.
    // If 'check_inten' is TRUE, then check the applicable PMINTENCLR_EL1 bit.
    // If 'exclude_cyc' is TRUE, then CCNTR is NOT checked.
    var check_e : boolean;
    var check_inten : boolean;
    var exclude_cyc : boolean;
    case reason of
        when PMUOverflowCondition PMUException =>
            check_e = TRUE;
            check_inten = TRUE;
            exclude_cyc = FALSE;
        when PMUOverflowCondition BRBEFreeze =>
            check_e = FALSE;
            check_inten = FALSE;
            exclude_cyc = TRUE;
        when PMUOverflowCondition Freeze =>
            check_e = FALSE;
            check_inten = FALSE;
            exclude_cyc = TRUE;
        when PMUOverflowCondition IRQ, PMUOverflowCondition EDBGRO =>
            check_e = TRUE;
            check_inten = TRUE;
            exclude_cyc = FALSE;
        otherwise =>
            unreachable;
    end;

    var ovsf : bits(64);

    if HaveAArch64() then
        ovsf = PMOVSSET_EL0();
        ovsf[63:33] = Zeros{31};
        if !IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
            ovsf[INSTRUCTION\_COUNTER\_ID] = '0';
        end;
    else
        ovsf = ZeroExtend{64}(PMOVSSET());
    end;

    let counters : integer{} = NUM_PMU_COUNTERS;
    // Remove unimplemented counters - these fields are RES0
    if counters < 31 then
        ovsf[30:counters] = Zeros{31-counters};
    end;

    for idx = 0 to counters - 1 do
        var global_en : bit;
        case GetPMUCounterRange(idx) of
            when PMUCounterRange R1 =>
                global_en = if HaveAArch64() then PMCR_EL0().E else PMCR().E;
                if !include_r1 then
                    ovsf[idx] = '0';
                end;
            when PMUCounterRange R2 =>
                global_en = if HaveAArch64() then MDCR_EL2().HPME else HDCR().HPME;
                if !include_r2 then
                    ovsf[idx] = '0';
                end;
            when PMUCounterRange R3 =>
                global_en = PMCCR().EPME;
                if !include_r3 then
                    ovsf[idx] = '0';
                end;
            otherwise =>
                unreachable;
        end;
        if check_e then
            ovsf[idx] = ovsf[idx] AND global_en;
        end;
    end;

    // Cycle counter
    if exclude_cyc || !include_r1 then
        ovsf[CYCLE\_COUNTER\_ID] = '0';
    end;
end;

```

```

if check_e then
    ovsf[CCYCLE COUNTER ID] = ovsf[CCYCLE COUNTER ID] AND PMCR_EL0().E;
end;

// Instruction counter
if HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
    if !include_r1 then
        ovsf[IINSTRUCTION COUNTER ID] = '0';
    end;
    if check_e then
        ovsf[IINSTRUCTION COUNTER ID] = ovsf[IINSTRUCTION COUNTER ID] AND PMCR_EL0().E;
    end;
end;

if check_inten then
    let inten : bits(64) = (if HaveAArch64() then PMINTENCLR_EL1()
                           else ZeroExtend{64}(PMINTENCLR()));
    ovsf = ovsf AND inten;
end;

return !IsZero(ovsf);
end;

```

Library pseudocode for shared/debug/pmu/ClearEventCounters

```

// ClearEventCounters()
// =====
// Zero all the event counters.
// Called on a write to PMCR_EL0 or PMCR that writes '1' to PMCR_EL0.P or PMCR.P.

func ClearEventCounters()
begin
    // Although ZeroPMUCounters implements the functionality for PMUACR_EL1
    // that is part of FEAT_PMUv3p9, it should be noted that writes to
    // PMCR_EL0 are not allowed at EL0 when PMUSERENR_EL0.UEN is 1, meaning
    // it is not relevant in this case.
    ZeroPMUCounters(Zeros{33} :: Ones{31});
end;

```



```

// CountPMUEvents()
// =====
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID (31).
// For the instruction counter, idx == INSTRUCTION_COUNTER_ID (32).

func CountPMUEvents(idx : integer) => boolean
begin
    let counters : integer = NUM_PMU_COUNTERS;
    assert (idx == CYCLE\_COUNTER\_ID || idx < counters ||
            (idx == INSTRUCTION\_COUNTER\_ID && IsFeatureImplemented(FEAT_PMuV3_ICNTR)));

    var debug : boolean;
    var enabled : boolean;
    var prohibited : boolean;
    var filtered : boolean;
    var frozen : boolean;

    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters
    let counter_range : PMUCounterRange = GetPMUCounterRange(idx);
    var ss : SecurityState = CurrentSecurityState();

    // Main enable controls
    var global_en : bit;
    var counter_en : bit;
    case counter_range of
        when PMUCounterRange\_R1 =>
            global_en = if HaveAArch64() then PMCR_EL0().E else PMCR().E;
        when PMUCounterRange\_R2 =>
            global_en = if HaveAArch64() then MDCR_EL2().HPME else HDCR().HPME;
        when PMUCounterRange\_R3 =>
            assert IsFeatureImplemented(FEAT_PMuV3_EXTPMN);
            global_en = PMCCR().EPME;
        otherwise =>
            unreachable;
    end;

    case idx of
        when INSTRUCTION\_COUNTER\_ID =>
            assert HaveAArch64();
            counter_en = PMCNTENSET_EL0().F0;
        when CYCLE\_COUNTER\_ID =>
            counter_en = if HaveAArch64() then PMCNTENSET_EL0().C else PMCNTENSET().C;
        otherwise =>
            counter_en = if HaveAArch64() then PMCNTENSET_EL0()[idx] else PMCNTENSET()[idx];
    end;

    enabled = global_en == '1' && counter_en == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state or at EL3 is prohibited if all of:
    // * EL3 is implemented
    // * One of the following is true:
    //   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
    //     - FEAT_PMuV3p7 is not implemented
    //     - MDCR_EL3.MPMX == 0
    //   - EL3 is using AArch32 and SDCR.SPME == 0
    // * Either not executing at EL0 using AArch32, or one of the following is true:
    //   - EL3 is using AArch32 and SDER.SUNIDEN == 0
    //   - EL3 is using AArch64, EL1 is using AArch32, and SDER32_EL3.SUNIDEN == 0
    // * PMNx is not reserved for use by the external interface
    if (HaveEL(EL3) && (ss == SS\_Secure || PSTATE.EL == EL3) &&
        counter_range != PMUCounterRange\_R3) then
        if ELUsingAArch32(EL3) then
            prohibited = (MDCR_EL3().SPME == '0' &&
                (!IsFeatureImplemented(FEAT_PMuV3p7) || MDCR_EL3().MPMX == '0'));
        else
            prohibited = SDCR().SPME == '0';
        end;

        if prohibited && PSTATE.EL == EL0 then
            if ELUsingAArch32(EL3) then
                prohibited = SDER().SUNIDEN == '0';
            elsif ELUsingAArch32(EL1) then
                prohibited = SDER32_EL3().SUNIDEN == '0';
            end;
        end;
    end;

    // Event counting at EL3 is prohibited if all of:
    // * FEAT_PMuV3p7 is implemented
    // * EL3 is using AArch64
    // * One of the following is true:
    //   - MDCR_EL3.SPME == 0

```



```

// - PMNx is not reserved for EL2
// * MDCR_EL3.MPMX == 1
// * PMNx is not reserved for use by the external interface
if (!prohibited && IsFeatureImplemented(FEAT_PMUv3p7) && PSTATE.EL == EL3 &&
    HaveAArch64() && counter_range != PMUCounterRange_R3) then
    prohibited = (MDCR_EL3().MPMX == '1' &&
        (MDCR_EL3().SPME == '0' || counter_range == PMUCounterRange_R1));
end;

// Event counting at EL2 is prohibited if all of:
// * FEAT_PMUv3p1 is implemented
// * PMNx is not reserved for EL2 or the external interface
// * EL2 is using AArch64 and MDCR_EL2.HPMD == 1, or EL2 is using AArch32 and HDCR.HPMD == 1
if (!prohibited && PSTATE.EL == EL2 && IsFeatureImplemented(FEAT_PMUv3p1) &&
    counter_range == PMUCounterRange_R1) then
    let hpmd : bit = if HaveAArch64() then MDCR_EL2().HPMD else HDCR().HPMD;
    prohibited = hpmd == '1';
end;

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !IsFeatureImplemented(FEAT_Debugv8p2) then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();
end;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if IsFeatureImplemented(FEAT_PMUv3p7) then
    var fz : bit;
    case counter_range of
        when PMUCounterRange_R1 =>
            fz = if HaveAArch64() then PMCR_EL0().FZO else PMCR().FZO;
        when PMUCounterRange_R2 =>
            fz = if HaveAArch64() then MDCR_EL2().HPMFZO else HDCR().HPMFZO;
        when PMUCounterRange_R3 =>
            fz = '0';
        otherwise =>
            unreachable;
    end;
    frozen = (fz == '1') && ShouldPMUFreeze(counter_range);
    frozen = frozen || SPEFreezeOnEvent(idx);
else
    frozen = FALSE;
end;

// PMCR_EL0.DP or PMCR.DP disables the cycle counter when event counting is prohibited
// or frozen
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    let dp : bit = if HaveAArch64() then PMCR_EL0().DP else PMCR().DP;
    enabled = enabled && dp == '0';
    // Otherwise whether event counting is prohibited or frozen does not affect the cycle
    // counter
    prohibited = FALSE;
    frozen = FALSE;
end;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR_EL0.DP.
if IsFeatureImplemented(FEAT_PMUv3p5) && idx == CYCLE_COUNTER_ID then
    if HaveEL(EL3) && (ss == SS_Secure || PSTATE.EL == EL3) then
        let sccd = if HaveAArch64() then MDCR_EL3().SCCD else SDCR().SCCD;
        if sccd == '1' then
            prohibited = TRUE;
        end;
    end;

    if PSTATE.EL == EL2 then
        let hccd = if HaveAArch64() then MDCR_EL2().HCCD else HDCR().HCCD;
        if hccd == '1' then
            prohibited = TRUE;
        end;
    end;
end;

// If FEAT_PMUv3p7 is implemented, cycle counting can be prohibited at EL3.
// This is not overridden by PMCR_EL0.DP.
if IsFeatureImplemented(FEAT_PMUv3p7) && idx == CYCLE_COUNTER_ID then
    if PSTATE.EL == EL3 && HaveAArch64() && MDCR_EL3().MCCD == '1' then
        prohibited = TRUE;
    end;
end;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH, RLK, RLU, RLH} bits
var filter : bits(32);
case idx of
    when INSTRUCTION_COUNTER_ID =>
        filter = PMICFILTR_EL0()[31:0];
    when CYCLE_COUNTER_ID =>
        filter = if HaveAArch64() then PMCCFILTR_EL0()[31:0] else PMCCFILTR();
    otherwise =>
        filter = if HaveAArch64() then PMEVTYPER_EL0(idx)[31:0] else PMEVTYPER(idx);
end;

```

```

end;

let p : bit = filter[31];
let u : bit = filter[30];
let nsk : bit = if HaveEL(EL3) then filter[29] else '0';
let nsu : bit = if HaveEL(EL3) then filter[28] else '0';
let nsh : bit = if HaveEL(EL2) then filter[27] else '0';
let m : bit = if HaveEL(EL3) && HaveAArch64() then filter[26] else '0';
let sh : bit = if HaveEL(EL3) && IsFeatureImplemented(FEAT_SEL2) then filter[24] else '0';
let rlk : bit = if IsFeatureImplemented(FEAT_RME) then filter[22] else '0';
let rlu : bit = if IsFeatureImplemented(FEAT_RME) then filter[21] else '0';
let rlh : bit = if IsFeatureImplemented(FEAT_RME) then filter[20] else '0';

ss = CurrentSecurityState();
case PSTATE.EL of
  when EL0 =>
    case ss of
      when SS_NonSecure => filtered = u != nsu;
      when SS_Secure => filtered = u == '1';
      when SS_Realm => filtered = u != rlu;
    end;
  when EL1 =>
    case ss of
      when SS_NonSecure => filtered = p != nsk;
      when SS_Secure => filtered = p == '1';
      when SS_Realm => filtered = p != rlk;
    end;
  when EL2 =>
    case ss of
      when SS_NonSecure => filtered = nsh == '0';
      when SS_Secure => filtered = nsh == sh;
      when SS_Realm => filtered = nsh == rlh;
    end;
  when EL3 =>
    if HaveAArch64() then
      filtered = m != p;
    else
      filtered = p == '1';
    end;
end;

end;

if IsFeatureImplemented(FEAT_PMUV3_SME) then
  let is_streaming_mode : boolean = PSTATE.SM == '1';
  var vs : bits(2);
  case idx of
    when INSTRUCTION_COUNTER_ID =>
      vs = PMICFILTR_EL0().VS;
    when CYCLE_COUNTER_ID =>
      vs = PMCCFILTR_EL0().VS;
    otherwise =>
      vs = PMEVTYPER_EL0(idx).VS;
  end;

  var streaming_mode_filtered : boolean;
  if vs == '11' then
    streaming_mode_filtered = ConstrainUnpredictableBool(Unpredictable_RES_PMU_VS);
  else
    streaming_mode_filtered = ((is_streaming_mode && vs[0] == '1') ||
      (!is_streaming_mode && vs[1] == '1'));
  end;

  filtered = filtered || streaming_mode_filtered;
end;

return !debug && enabled && !prohibited && !filtered && !frozen;
end;

```

Library pseudocode for shared/debug/pmu/EffectiveEPMN

```
// EffectiveEPMN()
// =====
// Returns the Effective value of PMCCR.EPMN.

func EffectiveEPMN() => bits(5)
begin
    let counters : integer = NUM_PMU_COUNTERS;
    var epmn_bits : bits(5);

    if IsFeatureImplemented(FEAT_PMuV3_EXTPMN) then
        epmn_bits = PMCCR().EPMN;
        if UInt(epmn_bits) > counters then
            (-, epmn_bits) = ConstrainUnpredictableBits{5} (Unpredictable RES EPMN);
        end;
    else
        epmn_bits = counters[4:0];
    end;

    return epmn_bits;
end;
```

Library pseudocode for shared/debug/pmu/EffectiveHPMN

```
// EffectiveHPMN()
// =====
// Returns the Effective value of MDCR_EL2.HPMN or HDCR.HPMN.

func EffectiveHPMN() => bits(5)
begin
    let counters : integer = UInt(EffectiveEPMN());
    var hpmn_bits : bits(5);

    if HaveEL(EL2) then // Software can reserve some event counters for EL2
        hpmn_bits = if HaveAArch64() then MDCR_EL2().HPMN else HDCR().HPMN;

        // When FEAT_PMuV3_EXTPMN is implemented, out of range values are capped.
        if UInt(hpmn_bits) > counters && IsFeatureImplemented(FEAT_PMuV3_EXTPMN) then
            hpmn_bits = counters[4:0];
        end;

        if (UInt(hpmn_bits) > counters ||
            (!IsFeatureImplemented(FEAT_HPMN0) && IsZero(hpmn_bits))) then
            (-, hpmn_bits) = ConstrainUnpredictableBits{5} (Unpredictable RES HPMN);
        end;
    else
        hpmn_bits = counters[4:0];
    end;

    return hpmn_bits;
end;
```

Library pseudocode for shared/debug/pmu/GetNumEventCountersAccessible

```
// GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current Exception level.

func GetNumEventCountersAccessible() => integer{0..NUM_PMU_COUNTERS}
begin
    var n : integer;

    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(EffectiveHPMN());
    else
        n = UInt(EffectiveEPMN());
    end;

    return n as integer{0..NUM_PMU_COUNTERS};
end;
```

Library pseudocode for shared/debug/pmu/GetNumEventCountersSelfHosted

```
// GetNumEventCountersSelfHosted()
// =====
// Return the number of event counters that can be accessed by the Self-hosted software.

func GetNumEventCountersSelfHosted() => integer
begin
  if IsFeatureImplemented(FEAT_PMUv3_EXTPMN) then
    return UInt(EffectiveEPMN());
  else
    return NUM_PMU_COUNTERS;
  end;
end;
```

Library pseudocode for shared/debug/pmu/GetPMUAccessMask

```
// GetPMUAccessMask()
// =====
// Return a mask of the PMU counters accessible at the current Exception level

func GetPMUAccessMask() => bits(64)
begin
  var mask : bits(64) = Zeros{};

  // PMICNTR_EL0 is only accessible at EL0 using AArch64 when PMUSERENR_EL0.UEN is 1.
  if IsFeatureImplemented(FEAT_PMUv3_ICNTR) && !UsingAArch32() then
    assert IsFeatureImplemented(FEAT_PMUv3p9);
    if PSTATE.EL != EL0 || PMUSERENR_EL0().UEN == '1' then
      mask[INSTRUCTION_COUNTER_ID] = '1';
    end;
  end;

  // PMCCNTR_EL0 is always implemented and accessible
  mask[CYCLE_COUNTER_ID] = '1';

  // PMEVCNTR_EL0(n)
  let counters : integer{} = GetNumEventCountersAccessible();
  if counters > 0 then
    mask[counters-1:0] = Ones(counters);
  end;

  // Check EL0 ignore access conditions
  if (IsFeatureImplemented(FEAT_PMUv3p9) && !ELUsingAArch32(EL1) &&
      PSTATE.EL == EL0 && PMUSERENR_EL0().UEN == '1') then
    mask = mask AND PMUACR_EL1(); // User access control
  end;

  return mask;
end;
```

Library pseudocode for shared/debug/pmu/GetPMUCounterRange

```
// GetPMUCounterRange()
// =====
// Returns the range that a counter is currently in.

func GetPMUCounterRange(n : integer) => PMUCounterRange
begin

  let counters : integer = NUM_PMU_COUNTERS;
  let epmn : integer = UInt(EffectiveEPMN());
  let hpmn : integer = UInt(EffectiveHPMN());

  if n < hpmn then
    return PMUCounterRange_R1;
  elsif n < epmn then
    return PMUCounterRange_R2;
  elsif n < counters then
    assert IsFeatureImplemented(FEAT_PMUv3_EXTPMN);
    return PMUCounterRange_R3;
  elsif n == CYCLE_COUNTER_ID then
    return PMUCounterRange_R1;
  elsif n == INSTRUCTION_COUNTER_ID then
    assert IsFeatureImplemented(FEAT_PMUv3_ICNTR);
    return PMUCounterRange_R1;
  else
    unreachable;
  end;
end;
```

Library pseudocode for shared/debug/pmu/GetPMUReadMask

```
// GetPMUReadMask()
// =====
// Return a mask of the PMU counters that can be read at the current
// Exception level.
// This mask masks reads from PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, and PMOVSCLR_EL0.

func GetPMUReadMask() => bits(64)
begin
    var mask : bits(64) = GetPMUAccessMask\(\);

    // Additional PMICNTR_EL0 accessibility checks. PMICNTR_EL0 controls read-as-zero
    // if a read of PMICFILTR_EL0 would be trapped to a higher Exception level.
    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) && mask[INSTRUCTION\_COUNTER\_ID] == '1' then
        // Check for trap to EL3.
        if HaveEL\(EL3\) && PSTATE.EL != EL3 && MDCR_EL3().EnPM2 == '0' then
            mask[INSTRUCTION\_COUNTER\_ID] = '0';
        end;

        // Check for trap to EL2.
        if EL2Enabled\(\) && PSTATE.EL IN {EL0, EL1} && !ELIsInHost\(EL0\) then
            // If FEAT_PMUv3_ICNTR and EL2 are implemented, then so is FEAT_FGT2.
            assert IsFeatureImplemented(FEAT_FGT2);
            if ((HaveEL\(EL3\) && SCR_EL3().FGTEn2 == '0') ||
                HDFGRTR2_EL2().nPMICFILTR_EL0 == '0') then
                mask[INSTRUCTION\_COUNTER\_ID] = '0';
            end;
        end;
    end;

    // Traps on other counters do not affect those counters' controls in the same way.

    return mask;
end;
```

Library pseudocode for shared/debug/pmu/GetPMUWriteMask

```
// GetPMUWriteMask()
// =====
// Return a mask of the PMU counters writable at the current Exception level.
// This mask masks writes to PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, PMOVSCCLR_EL0, and PMZR_EL0.
// 'write_counter' is TRUE for a write to PMZR_EL0, when the counter is being
// updated, and FALSE for other cases when the controls are being updated.

func GetPMUWriteMask(write_counter : boolean) => bits(64)
begin
    var mask : bits(64) = GetPMUAccessMask\(\);

    // Check EL0 ignore write conditions
    if (IsFeatureImplemented(FEAT_PMuV3p9) && !ELUsingAArch32\(EL1\) &&
        PSTATE.EL == EL0 && PMUSERENR_EL0().UEN == '1') then
        if (IsFeatureImplemented(FEAT_PMuV3_ICNTR) &&
            PMUSERENR_EL0().IR == '1') then // PMICNTR_EL0 read-only
            mask[INSTRUCTION\_COUNTER\_ID] = '0';
        end;
        if PMUSERENR_EL0().CR == '1' then // PMCCNTR_EL0 read-only
            mask[CYCLE\_COUNTER\_ID] = '0';
        end;
        if PMUSERENR_EL0().ER == '1' then // PMEVCNTR[n]_EL0 read-only
            mask[30:0] = Zeros{31};
        end;
    end;

    // Additional PMICNTR_EL0 accessibility checks. PMICNTR_EL0 controls ignore writes
    // if a write of PMICFILTR_EL0 would be trapped to a higher Exception level.
    // Indirect writes to PMICNTR_EL0 (through PMZR_EL0) are ignored if a write of
    // PMICNTR_EL0 would be trapped to a higher Exception level.
    if IsFeatureImplemented(FEAT_PMuV3_ICNTR) && mask[INSTRUCTION\_COUNTER\_ID] == '1' then
        // Check for trap to EL3.
        if HaveEL\(EL3\) && PSTATE.EL != EL3 && MDCR_EL3().EnPM2 == '0' then
            mask[INSTRUCTION\_COUNTER\_ID] = '0';
        end;

        // Check for trap to EL2.
        if EL2Enabled\(\) && PSTATE.EL IN {EL0, EL1} && !ELIsInHost\(EL0\) then
            // If FEAT_PMuV3_ICNTR and EL2 are implemented, then so is FEAT_FGT2.
            assert IsFeatureImplemented(FEAT_FGT2);
            let fgt_bit : bit = (if write_counter then HDBGWTR2_EL2().nPMICNTR_EL0
                                else HDBGWTR2_EL2().nPMICFILTR_EL0);
            if (HaveEL\(EL3\) && SCR_EL3().FGTEN2 == '0') || fgt_bit == '0' then
                mask[INSTRUCTION\_COUNTER\_ID] = '0';
            end;
        end;
    end;

    // Traps on other counters do not affect those counters' controls in the same way.

    return mask;
end;
```

Library pseudocode for shared/debug/pmu/HasElapsed64Cycles

```
// HasElapsed64Cycles()
// =====
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.

impdef func HasElapsed64Cycles() => boolean
begin
    return TRUE;
end;
```

Library pseudocode for shared/debug/pmu/IncrementInstructionCounter

```
// IncrementInstructionCounter()
// =====
// Increment the instruction counter and possibly set overflow bits.

func IncrementInstructionCounter(increment : integer)
begin
    if CountPMUEvents\(INSTRUCTION\_COUNTER\_ID\) then
        let old_value : integer = UInt(PMICNTR_EL0());
        let new_value : integer = old_value + increment;
        PMICNTR_EL0() = new_value[63:0];

        // The effective value of PMCR_EL0.LP is '1' for the instruction counter
        if old_value[64] != new_value[64] then
            PMOVSSET_EL0().FO = '1';
        end;
    end;
    return;
end;
```

Library pseudocode for shared/debug/pmu/IsRange3Counter

```
// IsRange3Counter()
// =====
// Returns TRUE if the counter is in the third range.

func IsRange3Counter(n : integer) => boolean
begin
    return PMUCounterRange\_R3 == GetPMUCounterRange(n);
end;
```

Library pseudocode for shared/debug/pmu/PMUCaptureEvent

```
// PMUCaptureEvent()
// =====
// If permitted and enabled, generate a PMU snapshot Capture event.

func PMUCaptureEvent()
begin
    assert HaveEL\(EL3\) && IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64\(\);
    let debug_state : boolean = Halted\(\);

    if !PMUCaptureEventAllowed\(\) then
        // Indicate a Capture event completed, unsuccessfully
        PMSSCR_EL1().[NC,SS] = '10';
        return;
    end;
    let counters : integer = NUM_PMU_COUNTERS;
    for idx = 0 to counters - 1 do
        PMEVCNTSVR_EL1(idx) = PMEVCNTR_EL0(idx) as PMEVCNTSVR_EL1_Type;
    end;
    PMCCNTSVR_EL1() = PMCCNTR_EL0() as PMCCNTSVR_EL1_Type;

    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
        PMICNTSVR_EL1() = PMICNTR_EL0() as PMICNTSVR_EL1_Type;
    end;

    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCSTL().SS == '1' then
        if pc\_sample.valid && !debug_state then
            SetPCSRActive\(\);
            SetPCSample\(\);
        else
            PMPCSR()[31:0] = Ones{32};
        end;
    end;

    if (IsFeatureImplemented(FEAT_BRBE) && BranchRecordAllowed\(PSTATE.EL\) &&
        BRBCR_EL1().FZPSS == '1' && (!HaveEL\(EL2\) || BRBCR_EL1().FZPSS == '1')) then
        BRBEFreeze\(\);
    end;

    // Indicate a successful Capture event
    PMSSCR_EL1().[NC,SS] = '00';
    if !debug_state || ConstrainUnpredictableBool\(Unpredictable\_PMUSNAPSHOTEVENT\) then
        PMUEvent(PMU_EVENT_PMU_SNAPSHOT);
    end;

    return;
end;
```

Library pseudocode for shared/debug/pmu/PMUCaptureEventAllowed

```
// PMUCaptureEventAllowed()
// =====
// Returns TRUE if PMU Capture events are allowed, and FALSE otherwise.

func PMUCaptureEventAllowed() => boolean
begin
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || !HaveAArch64() then
        return FALSE;
    end;

    if !PMUCaptureEventEnabled() || OSLockStatus() then
        return FALSE;
    elseif HaveEL(EL3) && MDCR_EL3().PMSSE != '01' then
        return MDCR_EL3().PMSSE == '11';
    elseif HaveEL(EL2) && MDCR_EL2().PMSSE != '01' then
        return MDCR_EL2().PMSSE == '11';
    else
        var pmsse_el1 : bits(2) = PMECR_EL1().SSE;
        if pmsse_el1 == '01' then // Reserved value
            var c : Constraint;
            (c, pmsse_el1) = ConstrainUnpredictableBits{2}(Unpredictable_RESPMSSE);
            assert c IN {Constraint DISABLED, Constraint UNKNOWN};
            if c == Constraint DISABLED then pmsse_el1 = '00'; end;
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        end;
        return pmsse_el1 == '11';
    end;
end;
```

Library pseudocode for shared/debug/pmu/PMUCaptureEventEnabled

```
// PMUCaptureEventEnabled()
// =====
// Returns TRUE if PMU Capture events are enabled, and FALSE otherwise.

func PMUCaptureEventEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || !HaveAArch64() then
        return FALSE;
    end;
    if HaveEL(EL3) && MDCR_EL3().PMSSE != '01' then
        return MDCR_EL3().PMSSE == '1x';
    elseif HaveEL(EL2) && ELUsingAArch32(EL2) then
        return FALSE;
    elseif HaveEL(EL2) && MDCR_EL2().PMSSE != '01' then
        return MDCR_EL2().PMSSE == '1x';
    elseif ELUsingAArch32(EL1) then
        return FALSE;
    else
        var pmsse_el1 : bits(2) = PMECR_EL1().SSE;
        if pmsse_el1 == '01' then // Reserved value
            var c : Constraint;
            (c, pmsse_el1) = ConstrainUnpredictableBits{2}(Unpredictable_RESPMSSE);
            assert c IN {Constraint DISABLED, Constraint UNKNOWN};
            if c == Constraint DISABLED then pmsse_el1 = '00'; end;
        end;
        // Otherwise the value returned by ConstrainUnpredictableBits must be
        // a non-reserved value
        return pmsse_el1 == '1x';
    end;
end;
```


Library pseudocode for shared/debug/pmu/PMUCountValue

```
// PMUCountValue()
// =====
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by.
// 'Vb' is the base value of the event that event counter 'n' is configured to count.
// 'Vm' is the value to increment event counter 'n-1' by if 'n' is odd, zero otherwise.

func PMUCountValue(n : integer, Vb : integer, Vm : integer) => integer
begin
    assert (n MOD 2) == 1 || Vm == 0;
    assert n < NUM_PMU_COUNTERS;

    if !IsFeatureImplemented(FEAT_PMuV3_TH) || !HaveAArch64() then
        return Vb;
    end;

    let TH : integer = UInt(PMEVTYPERS_EL0(n).TH);

    // Control register fields
    var tc : bits(3) = PMEVTYPERS_EL0(n).TC;
    var te : bit = '0';
    if IsFeatureImplemented(FEAT_PMuV3_EDGE) then
        te = PMEVTYPERS_EL0(n).TE;
    end;
    var tlc : bits(2) = '00';
    if IsFeatureImplemented(FEAT_PMuV3_TH2) && (n MOD 2) == 1 then
        tlc = PMEVTYPERS_EL0(n).TLC;
    end;

    // Check for reserved cases
    var c : Constraint;
    (c, tc, te, tlc) = ReservedPMUThreshold(n, tc, te, tlc);
    if c == Constraint_DISABLED then
        return Vb;
    end;
    // Otherwise the values returned by ReservedPMUThreshold must be defined values

    // Check if disabled. Note that this function will return the value of Vb when
    // the control register fields are all zero, even without this check.
    if tc == '000' && TH == 0 && te == '0' && tlc == '00' then
        return Vb;
    end;

    // Threshold condition
    var Ct : boolean;
    case tc[2:1] of
        when '00' => Ct = (Vb != TH);           // Disabled or not-equal
        when '01' => Ct = (Vb == TH);           // Equals
        when '10' => Ct = (Vb >= TH);           // Greater-than-or-equal
        when '11' => Ct = (Vb < TH);            // Less-than
    end;

    var Vn : integer;
    if te == '1' then
        // Edge condition
        let Cp : boolean = PMULastThresholdValue[n];
        var Ce : boolean;
        var Ve : integer;
        case tc[1:0] of
            when '10' => Ce = (Cp != Ct);       // Both edges
            when 'x1' => Ce = (!Cp && Ct);      // Single edge
            otherwise => unreachable;           // Covered by ReservedPMUThreshold
        end;
        case tlc of
            when '00' => Ve = (if Ce then 1 else 0);
            when '10' => Ve = (if Ce then Vm else 0);
            otherwise => unreachable;           // Covered by ReservedPMUThreshold
        end;
        Vn = Ve;
    else
        // Threshold condition
        var Vt : integer;
        case tc[0]::tlc of
            when '0 00' => Vt = (if Ct then Vb else 0);
            when '0 01' => Vt = (if Ct then Vb else Vm);
            when '0 10' => Vt = (if Ct then Vm else 0);
            when '1 00' => Vt = (if Ct then 1 else 0);
            when '1 01' => Vt = (if Ct then 1 else Vm);
            otherwise => unreachable;           // Covered by ReservedPMUThreshold
        end;
        Vn = Vt;
    end;

    PMULastThresholdValue[n] = Ct;

    return Vn;
end;
```

Library pseudocode for shared/debug/pmu/PMUCounterRange

```
// PMUCounterRange
// =====
// Enumerates the ranges to which an event counter belongs to.

type PMUCounterRange of enumeration {
    PMUCounterRange_R1,
    PMUCounterRange_R2,
    PMUCounterRange_R3
};
```

Library pseudocode for shared/debug/pmu/PMUEvent

```
// PMUEvent()
// =====
// Generate a PMU event. By default, increment by 1.
func PMUEvent(pmuevent : bits(16))
begin
    PMUEvent(pmuevent, 1);
end;

// PMUEvent()
// =====
// Accumulate a PMU Event.

func PMUEvent(pmuevent : bits(16), increment : integer)
begin
    let counters : integer = NUM_PMU_COUNTERS;
    if counters != 0 then
        for idx = 0 to counters - 1 do
            PMUEvent(pmuevent, increment, idx);
        end;
    end;

    if (HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) &&
        pmuevent == PMU_EVENT_INST_RETIRED) then
        IncrementInstructionCounter(increment);
    end;
end;

// PMUEvent()
// =====
// Accumulate a PMU Event for a specific event counter.

func PMUEvent(pmuevent : bits(16), increment : integer, idx : integer)
begin
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    end;
    if UsingAArch32() then
        if PMEVTYPER(idx).evtCount == pmuevent then
            PMUEventAccumulator[[idx]] = PMUEventAccumulator[[idx]] + increment;
        end;
    else
        if PMEVTYPER_ELO(idx).evtCount == pmuevent then
            PMUEventAccumulator[[idx]] = PMUEventAccumulator[[idx]] + increment;
        end;
    end;
end;
end;
```

Library pseudocode for shared/debug/pmu/PMUOverflowCondition

```
// PMUOverflowCondition()
// =====
// Enumerates the reasons for which the PMU overflow condition is evaluated.

type PMUOverflowCondition of enumeration {
    PMUOverflowCondition_PMUException,
    PMUOverflowCondition_BRBEFreeze,
    PMUOverflowCondition_Freeze,
    PMUOverflowCondition_IRQ,
    PMUOverflowCondition_EDBGREQ
};
```

Library pseudocode for shared/debug/pmu/PMUSwIncrement

```
// PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC

func PMUSwIncrement(sw_incr_in : bits(64))
begin
    var sw_incr : bits(64) = sw_incr_in;
    var mask : bits(31) = Zeros{};
    let counters : integer{} = GetNumEventCountersAccessible\(\);
    if counters > 0 then
        mask[counters-1:0] = Ones{counters};
    end;

    if (IsFeatureImplemented(FEAT_PMUv3p9) && !ELUsingAArch32\(EL1\) &&
        PSTATE.EL == EL0 && PMUSERENR_EL0().[UEN,SW] == '10') then
        mask = mask AND PMUACR_EL1()[30:0];
    end;

    sw_incr = sw_incr AND ZeroExtend{64}(mask);
    for idx = 0 to 30 do
        if sw_incr[idx] == '1' then
            PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
        end;
    end;

    return;
end;
```

Library pseudocode for shared/debug/pmu/ReservedPMUThreshold

```
// ReservedPMUThreshold()
// =====
// Checks if the given PMEVTYPER_EL1().[TH,TE,TLC] values are reserved and will
// generate Constrained Unpredictable behavior, otherwise return Constraint_NONE.

func ReservedPMUThreshold(n : integer, tc_in : bits(3),
                          te_in : bit, tlc_in : bits(2)) => (Constraint, bits(3), bit, bits(2))
begin
    var tc : bits(3) = tc_in;
    var te : bit = te_in;
    var tlc : bits(2) = tlc_in;

    var reserved : boolean = FALSE;

    if IsFeatureImplemented(FEAT_PMUv3_EDGE) then
        if te == '1' && tc[1:0] == '00' then // Edge condition
            reserved = TRUE;
        end;
    else
        te = '0'; // Control is RES0
    end;

    if IsFeatureImplemented(FEAT_PMUv3_TH2) && (n MOD 2) == 1 then
        if tlc == '11' then // Reserved value
            reserved = TRUE;
        end;
        if te == '1' then // Edge condition
            if tlc == '01' then
                reserved = TRUE;
            end;
        else // Threshold condition
            if tc[0] == '1' && tlc == '10' then
                reserved = TRUE;
            end;
        end;
    else
        tlc = '00'; // Controls are RES0
    end;

    var c : Constraint = Constraint\_NONE;
    if reserved then
        var unpred_reserved_bits : bits(6);
        (c, unpred_reserved_bits) = ConstrainUnpredictableBits{6}(Unpredictable\_RESTC);
        tc = unpred_reserved_bits[5:3];
        te = unpred_reserved_bits[2];
        tlc = unpred_reserved_bits[1:0];
    end;

    return (c, tc, te, tlc);
end;
```

Library pseudocode for shared/debug/pmu/SMEPMUEventPredicate

```
// SMEPMUEventPredicate()
// =====
// Call the relevant PMU predication events based on the SME instruction properties.

func SMEPMUEventPredicate(N)(mask1 : bits(N), mask2 : bits(N), esize : integer)
begin
    PMUEvent(PMU_EVENT_SVE_PRED_SPEC);
    PMUEvent(PMU_EVENT_SME_PRED2_SPEC);
    if AllElementsActive{N}(mask1, esize) && AllElementsActive{N}(mask2, esize) then
        PMUEvent(PMU_EVENT_SME_PRED2_FULL_SPEC);
    else
        PMUEvent(PMU_EVENT_SME_PRED2_NOT_FULL_SPEC);
        if !AnyActiveElement{N}(mask1, esize) && !AnyActiveElement{N}(mask2, esize) then
            PMUEvent(PMU_EVENT_SME_PRED2_EMPTY_SPEC);
        else
            PMUEvent(PMU_EVENT_SME_PRED2_PARTIAL_SPEC);
        end;
    end;
end;
```

Library pseudocode for shared/debug/pmu/SVEPMUEventPredicate

```
// SVEPMUEventPredicate()
// =====
// Call the relevant PMU predication events based on the SVE instruction properties.

func SVEPMUEventPredicate(N)(mask : bits(N), esize : integer)
begin
    PMUEvent(PMU_EVENT_SVE_PRED_SPEC);
    if AllElementsActive{N}(mask, esize) then
        PMUEvent(PMU_EVENT_SVE_PRED_FULL_SPEC);
    else
        PMUEvent(PMU_EVENT_SVE_PRED_NOT_FULL_SPEC);
        if !AnyActiveElement{N}(mask, esize) then
            PMUEvent(PMU_EVENT_SVE_PRED_EMPTY_SPEC);
        else
            PMUEvent(PMU_EVENT_SVE_PRED_PARTIAL_SPEC);
        end;
    end;
end;
```

Library pseudocode for shared/debug/pmu/ShouldPMUFreeze

```
// ShouldPMUFreeze()
// =====

func ShouldPMUFreeze(r : PMUCounterRange) => boolean
begin
    let include_r1 : boolean = (r == PMUCounterRange_R1);
    let include_r2 : boolean = (r == PMUCounterRange_R2);
    let include_r3 : boolean = FALSE;

    if r == PMUCounterRange_R3 then
        return FALSE;
    end;

    let overflow : boolean = CheckPMUOverflowCondition(PMUOverflowCondition_Freeze,
                                                         include_r1, include_r2, include_r3);

    return overflow;
end;
```

Library pseudocode for shared/debug/pmu/ZeroCycleCounter

```
// ZeroCycleCounter()
// =====
// Called on a write to PMCR_EL0 or PMCR that writes '1' to PMCR_EL0.C or PMCR.C.

func ZeroCycleCounter()
begin
    var mask : bits(64) = Zeros{};
    mask[CYCLE_COUNTER_ID] = '1';
    ZeroPMUCounters(mask);
end;
```

Library pseudocode for shared/debug/pmu/ZeroPMUCounters

```
// ZeroPMUCounters()
// =====
// Zero set of counters specified by the mask in 'val'.
// For a System register write to PMZR_EL0, 'val' is the value passed in X<t>.

func ZeroPMUCounters(val : bits(64))
begin
    let masked_val : bits(64) = val AND GetPMUWriteMask(TRUE);

    for idx = 0 to 63 do
        if masked_val[idx] == '1' && !IsRange3Counter(idx) then
            case idx of
                when INSTRUCTION\_COUNTER\_ID =>
                    PMICNTR_EL0() = Zeros{64};
                when CYCLE\_COUNTER\_ID =>
                    if !HaveAArch64() then
                        PMCCNTR() = Zeros{64};
                    else
                        PMCCNTR_EL0() = Zeros{64};
                    end;
                otherwise =>
                    if !HaveAArch64() then
                        PMEVCNTR(idx) = Zeros{32};
                    elsif IsFeatureImplemented(FEAT_PMUv3p5) then
                        PMEVCNTR_EL0(idx) = Zeros{64};
                    else
                        PMEVCNTR_EL0(idx)[31:0] = Zeros{32};
                    end;
            end;
        end;
    end;

    return;
end;
```

Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// =====

func CreatePCSample()
begin
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.
    if PCSRSSuspended() then return; end;
    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr{64}();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = CurrentSecurityState();
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR() else CONTEXTIDR_EL1()[31:0];
    pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

    if pc_sample.has_el2 then
        pc_sample.vmid = VMID();
        if ((IsFeatureImplemented(FEAT_VHE) || IsFeatureImplemented(FEAT_Debugv8p2)) &&
            !ELUsingAArch32(EL2)) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2()[31:0];
        else
            pc_sample.contextidr_el2 = ARBITRARY : bits(32);
        end;
        pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
    end;
    return;
end;
```

Library pseudocode for shared/debug/samplebasedprofiling/PCSRSSuspended

```
// PCSRSSuspended()
// =====
// Returns TRUE if PC Sample-based Profiling is suspended, and FALSE otherwise.

func PCSRSSuspended() => boolean
begin
    if IsFeatureImplemented(FEAT_PMUv3_SS) && PMPCSTL().SS == '1' then
        return FALSE;
    end;
    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCSTL().IMP == '1' then
        return PMPCSTL().EN == '0';
    end;
    return ImpDefBool("PCSR is suspended");
end;
```

Library pseudocode for shared/debug/samplebasedprofiling/PCSample

```
// PCSample
// =====

type PCSample of record {
    valid : boolean,
    pc : bits(64),
    el : bits(2),
    rw : bit,
    ss : SecurityState,
    has_el2 : boolean,
    contextidr : bits(32),
    contextidr_el2 : bits(32),
    el0h : boolean,
    vmid : bits(NUM_VMIDBITS)
};

var pc_sample : PCSample;
```

Library pseudocode for shared/debug/samplebasedprofiling/Read_EDPCSRlo

```
// Read_EDPCSRlo()
// =====

func Read_EDPCSRlo(memory_mapped : boolean) => bits(32)
begin
    // The Software lock is OPTIONAL.
    let update : boolean = !memory_mapped || EDLSR().SLK == '0'; // Software locked: no side-effects
    var sample : bits(32);
    if pc\_sample.valid then
        sample = pc\_sample.pc[31:0];
        if update then
            if IsFeatureImplemented(FEAT_VHE) && EDSCR().SC2 == '1' then
                EDPCRhi.PC = (if pc\_sample.rw == '0' then Zeros{24} else pc\_sample.pc[55:32]);
                EDPCRhi.EL = pc\_sample.el;
                EDPCRhi.NS = (if pc\_sample.ss == SS Secure then '0' else '1');
            else
                EDPCRhi = (if pc\_sample.rw == '0' then Zeros{32} else pc\_sample.pc[63:32]);
            end;
            EDCIDSR() = pc\_sample.contextidr;
            if ((IsFeatureImplemented(FEAT_VHE) || IsFeatureImplemented(FEAT_Debugv8p2)) &&
                EDSCR().SC2 == '1') then
                EDVIDSR() = (if pc\_sample.has_el2 then pc\_sample.contextidr_el2
                    else ARBITRARY : bits(32));
            else
                EDVIDSR().VMID = (if pc\_sample.has_el2 && pc\_sample.el IN {EL1,EL0}
                    then pc\_sample.vmid else Zeros{NUM_VMIDBITS});
                EDVIDSR().NS = (if pc\_sample.ss == SS Secure then '0' else '1');
                EDVIDSR().E2 = (if pc\_sample.el == EL2 then '1' else '0');
                EDVIDSR().E3 = (if pc\_sample.el == EL3 then '1' else '0') AND pc\_sample.rw;
                // The conditions for setting HV are not specified if PCRhi is zero.
                // An example implementation may be "pc_sample.rw".
                EDVIDSR().HV = (if !IsZero(EDPCRhi) then '1'
                    else ImpDefBit("0 or 1"));
            end;
        end;
    else
        sample = Ones{32};
        if update then
            EDPCRhi = ARBITRARY : bits(32);
            EDCIDSR() = ARBITRARY : bits(32);
            EDVIDSR() = ARBITRARY : bits(32);
        end;
    end;

    return sample;
end;
```

Library pseudocode for shared/debug/samplebasedprofiling/Read_PMPCSR

```
// Read_PMPCSR()
// =====

func Read_PMPCSR(memory_mapped : boolean) => bits(64)
begin
    // The Software lock is OPTIONAL.
    var update : boolean = !memory_mapped || PMLSR().SLK == '0'; // Software locked: no side-effects

    if IsFeatureImplemented(FEAT_PCSRv8p9) && update then
        if IsFeatureImplemented(FEAT_PMUv3_SS) && PMPCSTL().SS == '1' then
            update = FALSE;
        elsif PMPCSTL().[IMP,EN] == '10' || (PMPCSTL().IMP == '0' && PCSRsuspended\(\)) then
            pc\_sample.valid = FALSE;
            SetPCSRActive\(\);
        end;
    end;

    if pc\_sample.valid then
        if update then SetPCSample\(\); end;
        return PMPCSR();
    else
        if update then SetPCSRUnknown\(\); end;
        return (ARBITRARY : bits(32) :: Ones{32});
    end;
end;
```

Library pseudocode for shared/debug/samplebasedprofiling/SetPCSRActive

```
// SetPCSRActive()
// =====
// Sets PC Sample-based Profiling to active state.

func SetPCSRActive()
begin
    if PMPCSTL().IMP == '1' then
        PMPCSTL().EN = '1';
    // If PMPCSTL.IMP reads as `0b0`, then PMPCSTL.EN is RES0, and it is
    // IMPLEMENTATION DEFINED whether PSCR is suspended or active at reset.
    end;
end;
```

Library pseudocode for shared/debug/samplebasedprofiling/SetPCSRUnknown

```
// SetPCSRUnknown()
// =====
// Sets the PC sample registers to UNKNOWN values because PC sampling
// is prohibited.

func SetPCSRUnknown()
begin
    PMPCSR()[31:0] = Ones{32};
    PMPCSR()[55:32] = ARBITRARY : bits(24);
    PMPCSR().EL = ARBITRARY : bits(2);
    PMPCSR().NS = ARBITRARY : bit;

    if IsFeatureImplemented(FEAT_PMUv3_EXT64) then
        PMCCIDSR() = ARBITRARY : bits(64);
        PMVCIDSR().VMID = ARBITRARY : bits(NUM_VMIDBITS);
    end;
    if IsFeatureImplemented(FEAT_PMUv3_EXT32) then
        PMCID1SR() = ARBITRARY : bits(32);
        PMCID2SR() = ARBITRARY : bits(32);
        PMVIDSR().VMID = ARBITRARY : bits(NUM_VMIDBITS);
    end;

    return;
end;
```

Library pseudocode for shared/debug/samplebasedprofiling/SetPCSample

```
// SetPCSample()
// =====
// Sets the PC sample registers to the appropriate sample values.

func SetPCSample()
begin
    PMPCSR()[31:0] = pc\_sample.pc[31:0];
    PMPCSR()[55:32] = (if pc\_sample.rw == '0' then Zeros{24} else pc\_sample.pc[55:32]);
    PMPCSR().EL = pc\_sample.el;
    if IsFeatureImplemented(FEAT_RME) then
        case pc\_sample.ss of
            when SS Secure =>
                PMPCSR().NSE = '0'; PMPCSR().NS = '0';
            when SS NonSecure =>
                PMPCSR().NSE = '0'; PMPCSR().NS = '1';
            when SS Root =>
                PMPCSR().NSE = '1'; PMPCSR().NS = '0';
            when SS Realm =>
                PMPCSR().NSE = '1'; PMPCSR().NS = '1';
        end;
    else
        PMPCSR().NS = (if pc\_sample.ss == SS Secure then '0' else '1');
    end;
    let contextidr_el2 : bits(32) = (if pc\_sample.has\_el2 then
        pc\_sample.contextidr\_el2 else ARBITRARY : bits(32));
    var vmid : bits(NUM_VMIDBITS) = ARBITRARY : bits(NUM_VMIDBITS);
    if pc\_sample.has\_el2 && pc\_sample.el IN {EL1,EL0} && !pc\_sample.el0h then
        vmid = pc\_sample.vmid;
    end;
    if IsFeatureImplemented(FEAT_PMUv3_EXT64) then
        PMCCIDSR() = contextidr_el2::pc\_sample.contextidr;
        PMVCIDSR().VMID = vmid;
    end;
    if IsFeatureImplemented(FEAT_PMUv3_EXT32) then
        PMCID1SR() = pc\_sample.contextidr;
        PMCID2SR() = contextidr_el2;
        PMVIDSR().VMID = vmid;
    end;

    return;
end;
```

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// =====
// Take a Software Step exception if in the active-pending state

func CheckSoftwareStep()
begin
    // Other self-hosted debug functions will call AArch32_GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64_GenerateDebugExceptions().
    let step_enabled : boolean = (!ELUsingAArch32\(DebugTarget\(\)\)) &&
        AArch64\_GenerateDebugExceptions\(\) &&
        MDSCR_EL1().SS == '1';
    let active_pending : boolean = step_enabled && PSTATE.SS == '0'; // active-pending
    if active_pending then
        AArch64\_SoftwareStepException\(\);
    end;
    ShouldAdvanceSS = TRUE;
    return;
end;
```


Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// =====
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

func DebugExceptionReturnSS{N}(spsr : bits(N)) => bit
begin
    assert Halted() || Restarting() || PSTATE.EL != EL0;

    var enabled_at_source : boolean;
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32_GenerateDebugExceptions();
    else
        enabled_at_source = AArch64_GenerateDebugExceptions();
    end;
    var valid : boolean;
    var dest_el : bits(2);
    if IllegalExceptionReturn{N}(spsr) then
        dest_el = PSTATE.EL;
    else
        (valid, dest_el) = ELFromSPSR{N}(spsr); assert valid;
    end;

    let dest_ss : SecurityState = SecurityStateAtEL(dest_el);
    var enabled_at_dest : boolean;
    let dest_using_32 : boolean = (if dest_el == EL0 then spsr[4] == '1'
                                   else ELUsingAArch32(dest_el));
    if dest_using_32 then
        enabled_at_dest = AArch32_GenerateDebugExceptionsFrom(dest_el, dest_ss);
    else
        let mask : bit = spsr[9];
        enabled_at_dest = AArch64_GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);
    end;

    let ELd : bits(2) = DebugTargetFrom(dest_ss);
    var SS_bit : bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1().SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr[21];
    else
        SS_bit = '0';
    end;
    return SS_bit;
end;
```

Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// =====
// Advance the Software Step state machine.

func SSAdvance()
begin
    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // PE only needs to consider advancing the state machine from the active-not-pending
    // state.
    if !ShouldAdvanceSS then return; end;
    let target : bits(2) = DebugTarget();
    let step_enabled : boolean = !ELUsingAArch32(target) && MDSCR_EL1().SS == '1';
    let active_not_pending : boolean = step_enabled && PSTATE.SS == '1';
    if active_not_pending then PSTATE.SS = '0'; end;
    ShouldAdvanceSS = FALSE;
    return;
end;
```

Library pseudocode for shared/debug/softwarestep/SoftwareStepOpEnabled

```
// SoftwareStepOpEnabled()
// =====
// Returns a boolean indicating if execution from MDSTEPOP_EL1 is enabled.

func SoftwareStepOpEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_STEP2) || UsingAArch32\(\) then
        return FALSE;
    end;

    let step_enabled = AArch64\_GenerateDebugExceptions\(\) && MDCR_EL1().SS == '1';
    let active_not_pending = step_enabled && PSTATE.SS == '1';
    let stepop = (MDCR_EL1().EnSTEPOP == '1' &&
        (!HaveEL\(EL3\) || MDCR_EL3().EnSTEPOP == '1') &&
        (!EL2Enabled\(\) || MDCR_EL2().EnSTEPOP == '1'));
    return active_not_pending && stepop;
end;
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_DidNotStep

```
// SoftwareStep_DidNotStep()
// =====
// Returns TRUE if the previously executed instruction was executed in the
// inactive state, that is, if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was an ISB
// or ERET executed in the active-not-pending state, or if another exception
// was taken before the Software Step exception. Returns FALSE otherwise,
// indicating that the previously executed instruction was executed in the
// active-not-pending state, that is, the instruction was stepped.

func SoftwareStep_DidNotStep() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_SteppedEX

```
// SoftwareStep_SteppedEX()
// =====
// Returns a value that describes the previously executed instruction. The
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX
// that failed its condition code test. Otherwise returns TRUE if the
// instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.

func SoftwareStep_SteppedEX() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/debug/watchpoint/DataCacheWatchpointSize

```
// DataCacheWatchpointSize()
// =====
// Return the IMPLEMENTATION DEFINED data cache watchpoint size
func DataCacheWatchpointSize() => integer
begin
    let size : integer = ImpDefInt("Data Cache Invalidate Watchpoint Size");
    assert IsPow2(size) && size >= 2^(UInt(CTR_EL0().DminLine) + 2) && size <= 2048;
    return size;
end;
```

Library pseudocode for shared/debug/watchpoint/WatchpointInfo

```
// WatchpointInfo
// =====
// Watchpoint related fields.

type WatchpointInfo of record {
    wptype : WatchpointType, // Type of watchpoint matched
    maybe_false_match : boolean, // Watchpoint matches rounded range
    watchpt_num : integer, // Matching watchpoint number
    value_match : boolean, // Watchpoint match
    vaddress : bits(64) // Matching Virtual Address
};
```

Library pseudocode for shared/debug/watchpoint/WatchpointType

```
// WatchpointType
// =====

type WatchpointType of enumeration {
    WatchpointType_Inactive,      // Watchpoint inactive or disabled
    WatchpointType_AddrMatch,     // Address Match watchpoint
    WatchpointType_AddrMismatch  // Address Mismatch watchpoint
};
```

Library pseudocode for shared/exceptions/aborts/EffectiveHCRX_EL2_TMEA

```
// EffectiveHCRX_EL2_TMEA()
// =====
// Return the Effective value of HCRX_EL2.TMEA.

func EffectiveHCRX_EL2_TMEA() => bit
begin
    if (IsFeatureImplemented(FEAT_DoubleFault2) && EL2Enabled() &&
        !ELUsingAArch32(EL2) && IsHCRXEL2Enabled()) then
        return HCRX_EL2().TMEA;
    else
        return '0';
    end;
end;
```

Library pseudocode for shared/exceptions/aborts/EffectiveHCR_AMO

```
// EffectiveHCR_AMO()
// =====
// Return the Effective value of HCR_EL2.AMO.

func EffectiveHCR_AMO() => bit
begin
    if EffectiveTGE() == '1' then
        return (if ELUsingAArch32(EL2) || EffectiveHCR_EL2_E2H() == '0' then '1' else '0');
    elsif EL2Enabled() then
        return (if ELUsingAArch32(EL2) then HCR().AMO else HCR_EL2().AMO);
    else
        return '0';
    end;
end;
```

Library pseudocode for shared/exceptions/aborts/EffectiveHCR_TEA

```
// EffectiveHCR_TEA()
// =====
// Return the Effective value of HCR_EL2.TEA.

func EffectiveHCR_TEA() => bit
begin
    if EL2Enabled() && IsFeatureImplemented(FEAT_RAS) then
        return (if ELUsingAArch32(EL2) then HCR2().TEA else HCR_EL2().TEA);
    else
        return '0';
    end;
end;
```

Library pseudocode for shared/exceptions/aborts/EffectiveNMEA

```
// EffectiveNMEA()
// =====
// Return the Effective value of SCR_EL3.NMEA or SCTLR2_ELx.NMEA.

func EffectiveNMEA() => bit
begin
    if IsFeatureImplemented(FEAT_DoubleFault2) then
        if PSTATE.EL == EL3 && !UsingAArch32() then
            return SCR_EL3().NMEA;
        elsif (PSTATE.EL == EL2 || IsInHost()) && !ELUsingAArch32(EL2) then
            return (if IsSCTLR2EL2Enabled() then SCTLR2_EL2().NMEA else '0');
        elsif !ELUsingAArch32(EL1) then
            return (if IsSCTLR2EL1Enabled() then SCTLR2_EL1().NMEA else '0');
        else
            return '0';
        end;
    elsif IsFeatureImplemented(FEAT_DoubleFault) && PSTATE.EL == EL3 && !UsingAArch32() then
        return SCR_EL3().NMEA AND EffectiveEA();
    else
        return '0';
    end;
end;
```

Library pseudocode for shared/exceptions/aborts/EffectiveSCR_EL3_TMEA

```
// EffectiveSCR_EL3_TMEA()
// =====
// Return the Effective value of SCR_EL3.TMEA.

func EffectiveSCR_EL3_TMEA() => bit
begin
    if (IsFeatureImplemented(FEAT_DoubleFault2) && HaveEL\(EL3\) &&
        !ELUsingAArch32\(EL3\)) then
        return SCR_EL3().TMEA;
    else
        return '0';
    end;
end;
```



```

// PhysicalSErrorTarget()
// =====
// Returns a tuple of whether SError exception can be taken and, if so, the
// target Exception level.
// If EL3 is implemented and using AArch32, then a target Exception level of
// EL1 means Abort mode, and EL3 means Monitor mode, including in Secure
// state when Abort mode is part of EL3.

func PhysicalSErrorTarget() => (boolean, bits(2))
begin
    if Halted() then
        return (TRUE, ARBITRARY : bits(2));
    end;

    let effective_ea : bit = EffectiveEA();
    let effective_amo : bit = EffectiveHCR_AMO();
    let effective_tge : bit = EffectiveTGE();
    let effective_nmea : bit = EffectiveNMEA();

    // When EL3 is implemented and using AArch32, the SCR.AW bit can allow PSTATE.A
    // to mask SError exceptions in Non-secure state when SCR.EA is 1 and the Effective
    // value of HCR.AMO is 0.
    var effective_aw : bit;
    if (ELUsingAArch32(EL3) && effective_ea == '1' &&
        CurrentSecurityState() == SS_NonSecure && effective_amo == '0') then
        effective_aw = SCR().AW;
    else
        effective_aw = '0';
    end;

    // The exception is masked by software.
    var masked : boolean;
    case PSTATE.EL of
        when EL3 =>
            masked = (!UsingAArch32() && effective_ea == '0') || PSTATE.A == '1';
        when EL2 =>
            masked = ((effective_ea == '0' || effective_aw == '1') &&
                ((!UsingAArch32() && effective_tge == '0' && effective_amo == '0') ||
                PSTATE.A == '1'));
        when EL1, EL0 =>
            masked = ((effective_ea == '0' || effective_aw == '1') &&
                effective_amo == '0' && PSTATE.A == '1');
    end;

    // When FEAT_DoubleFault or FEAT_DoubleFault2 is implemented, the mask might be overridden.
    masked = (masked && effective_nmea == '0');

    // External debug might disable the exception in the current Security state.
    // This is not relevant at EL3.
    let intdis : boolean = PSTATE.EL != EL3 && ExternalDebugInterruptsDisabled(EL1);

    var target_el : bits(2) = ARBITRARY : bits(2);
    if effective_ea == '1' || (PSTATE.EL == EL3 && !ELUsingAArch32(EL3)) then
        if !masked then target_el = EL3; end;

    elsif EL2Enabled() && effective_amo == '1' && !intdis && PSTATE.EL IN {EL0, EL1} then
        target_el = EL2;
        masked = FALSE;

    elsif (EffectiveHCRX_EL2_TMEA() == '1' && !intdis &&
        ((PSTATE.EL == EL1 && PSTATE.A == '1') ||
        (PSTATE.EL == EL0 && masked && !IsInHost()))) then
        target_el = EL2;
        masked = FALSE;

    elsif (EffectiveSCR_EL3_TMEA() == '1' &&
        ((PSTATE.EL IN {EL2, EL1} && PSTATE.A == '1') ||
        (PSTATE.EL IN {EL2, EL0} && masked) || intdis)) then
        target_el = EL3;
        masked = FALSE;

    elsif PSTATE.EL == EL2 || IsInHost() then
        if !masked then target_el = EL2; end;

    else
        assert (PSTATE.EL == EL1 ||
            (PSTATE.EL == EL3 && ELUsingAArch32(EL3)) ||
            (PSTATE.EL == EL0 && !IsInHost()));
        if !masked then target_el = EL1; end;
    end;

    // External debug might disable the exception for the target Exception level.
    if !masked && ExternalDebugInterruptsDisabled(target_el) then
        masked = TRUE;
        target_el = ARBITRARY : bits(2);
    end;

    return (masked, target_el);
end;

```

Library pseudocode for shared/exceptions/aborts/SyncExternalAbortTarget

```
// SyncExternalAbortTarget()
// =====
// Returns the target Exception level for a Synchronous External Data or
// Instruction or Prefetch Abort.
// If EL3 is implemented and using AArch32, then a target Exception level of
// EL1 means Abort mode, and EL3 means Monitor mode, including in Secure
// state when Abort mode is part of EL3.

func SyncExternalAbortTarget(fault : FaultRecord) => bits(2)
begin
    let effective_ea : bit = EffectiveEA();
    let effective_tea : bit = EffectiveHCR\_TEA();
    let effective_tge : bit = EffectiveTGE();

    var target_el : bits(2);
    if effective_ea == '1' || (PSTATE.EL == EL3 && !ELUsingAArch32(EL3)) then
        target_el = EL3;

    elsif (EL2Enabled() && PSTATE.EL IN {EL1, EL0} &&
        (effective_tea == '1' || IsSecondStage(fault) ||
        fault.accessdesc.acctype == AccessType\_NV2 ||
        (PSTATE.EL == EL0 && effective_tge == '1')))) then
        target_el = EL2;

    elsif EffectiveHCRX\_EL2\_TMEA() == '1' && PSTATE.A == '1' && PSTATE.EL == EL1 then
        target_el = EL2;

    elsif EffectiveSCR\_EL3\_TMEA() == '1' && PSTATE.A == '1' && PSTATE.EL IN {EL1, EL2} then
        target_el = EL3;

    else
        assert PSTATE.EL != EL3 || ELUsingAArch32(EL3);
        target_el = (if PSTATE.EL == EL2 then EL2 else EL1);
    end;

    return target_el;
end;
```

Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// =====
// Return CV and COND fields of instruction syndrome

func ConditionSyndrome() => bits(5)
begin
    var syndrome : bits(5);

    if UsingAArch32() then
        let cond = CurrentCond();
        if PSTATE.T == '0' then // A32
            syndrome[4] = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable\_ESRCONDPASS) then
                syndrome[3:0] = '1110';
            else
                syndrome[3:0] = cond;
            end;
        else // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            // * CV set to 0 and COND is set to an UNKNOWN value
            // * CV set to 1 and COND is set to the condition code for the condition that
            // applied to the instruction.
            if ImpDefBool("Condition valid for trapped T32") then
                syndrome[4] = '1';
                syndrome[3:0] = cond;
            else
                syndrome[4] = '0';
                syndrome[3:0] = ARBITRARY : bits(4);
            end;
        end;
    else
        syndrome[4] = '1';
        syndrome[3:0] = '1110';
    end;
    return syndrome;
end;
```

Library pseudocode for shared/exceptions/exceptions/Exception

```
// Exception
// =====
// Classes of exception.

type Exception of enumeration {
    Exception_Uncategorized, // Uncategorized or unknown reason
    Exception_WFxTrap,       // Trapped WFI or WFE instruction
    Exception_CPl5RTTTrap,   // Trapped AArch32 MCR or MRC access, coproc=0b111
    Exception_CPl5RRTTrap,   // Trapped AArch32 MCRR or MRRC access, coproc=0b1111
    Exception_CPl4RTTTrap,   // Trapped AArch32 MCR or MRC access, coproc=0b1110
    Exception_CPl4DTTTrap,   // Trapped AArch32 LDC or STC access, coproc=0b1110
    Exception_CPl4RRTTrap,   // Trapped AArch32 MRRC access, coproc=0b1110
    Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
    Exception_FPIDTrap,      // Trapped access to SIMD or FP ID register
    Exception_OtherTrap,     // Trapped access to other instructions
    // Trapped BXJ instruction not supported in Armv8
    Exception_PACTrap,       // Trapped invalid PAC use
    Exception_IllegalState,  // Illegal Execution state
    Exception_SupervisorCall, // Supervisor Call
    Exception_HypervisorCall, // Hypervisor Call
    Exception_MonitorCall,   // Monitor Call or Trapped SMC instruction
    Exception_SystemRegisterTrap, // Trapped MRS or MSR System register access
    Exception_ERetTrap,      // Trapped invalid ERET use
    Exception_InstructionAbort, // Instruction Abort or Prefetch Abort
    Exception_PCAalignment,   // PC alignment fault
    Exception_DataAbort,      // Data Abort
    Exception_NV2DataAbort,   // Data abort at EL1 reported as being from EL2
    Exception_PACFail,        // PAC Authentication failure
    Exception_SPAlignment,    // SP alignment fault
    Exception_FPTrappedException, // IEEE trapped FP exception
    Exception_SError,         // SError interrupt
    Exception_Breakpoint,     // (Hardware) Breakpoint
    Exception_SoftwareStep,   // Software Step
    Exception_Watchpoint,     // Watchpoint
    Exception_NV2Watchpoint,  // Watchpoint at EL1 reported as being from EL2
    Exception_SoftwareBreakpoint, // Software Breakpoint Instruction
    Exception_VectorCatch,    // AArch32 Vector Catch
    Exception_IRQ,            // IRQ interrupt
    Exception_SVEAccessTrap,  // HCPTR trapped access to SVE
    Exception_SMEAccessTrap,  // HCPTR trapped access to SME
    Exception_GPC,            // Granule protection check
    Exception_BranchTarget,   // Branch Target Identification
    Exception_MemCpyMemSet,   // Exception from a CPY* or SET* instruction
    Exception_GCSFail,        // GCS Exceptions
    Exception_Profiling,      // Profiling exception
    Exception_SystemRegister128Trap, // Trapped MRRS or MSRR System register or SYSP access
    Exception_FIQ;           // FIQ interrupt
```

Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```
// ExceptionRecord
// =====

type ExceptionRecord of record {
    excepttype : Exception, // Exception class
    syndrome : IssType,       // Syndrome record
    paddress : FullAddress,   // Physical fault address
    vaddress : bits(64),      // Virtual fault address
    ipavalid : boolean,       // Validity of Intermediate Physical fault address
    pavalid : boolean,        // Validity of Physical fault address
    NS : bit,                 // Intermediate Physical fault address space
    ipaddress : bits(NUM\_PABITS), // Intermediate Physical fault address
    trappedsyscallinst : boolean; // Trapped SVC or SMC instruction
```


Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// =====
// Return a blank exception syndrome record for an exception of the given type.

func ExceptionSyndrome(exceptype : Exception) => ExceptionRecord
begin
    var r : ExceptionRecord;

    r.exceptype      = exceptype;

    // Initialize all other fields
    r.syndrome.iss    = Zeros{25};
    r.syndrome.iss2   = Zeros{24};
    r.vaddress        = Zeros{64};
    r.ipavalid        = FALSE;
    r.pavalid         = FALSE;
    r.NS              = '0';
    r.ipaddress        = Zeros{NUM_PABITS};
    r.paddress.paspace = ARBITRARY : PASpace;
    r.paddress.address = ARBITRARY : bits(NUM\_PABITS);
    r.trappedsyscallinst = FALSE;
    return r;
end;
```

Library pseudocode for shared/exceptions/traps/Undefined

```
// Undefined()
// =====

noreturn func Undefined()
begin
    if UsingAArch32() then
        AArch32_Undefined();
    else
        AArch64_Undefined();
    end;
end;
```

Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =====
// Function that gives the Long-descriptor FSC code for types of Fault

func EncodeLDFSC(statuscode : Fault, level : integer) => bits(6)
begin
    var result : bits(6);

    // 128-bit descriptors will start from level -2 for 4KB to resolve bits IA[55:51]
    if level == -2 then
        assert IsFeatureImplemented(FEAT_D128);
        case statuscode of
            when Fault AddressSize => result = '101100';
            when Fault Translation => result = '101010';
            when Fault SyncExternalOnWalk => result = '010010';
            when Fault SyncParityOnWalk =>
                result = '011010';
                assert !IsFeatureImplemented(FEAT_RAS);
            when Fault GPCFOnWalk => result = '100010';
            otherwise => unreachable;
        end;
        return result;
    end;

    if level == -1 then
        assert IsFeatureImplemented(FEAT_LPA2);
        case statuscode of
            when Fault AddressSize => result = '101001';
            when Fault Translation => result = '101011';
            when Fault SyncExternalOnWalk => result = '010011';
            when Fault SyncParityOnWalk =>
                result = '011011';
                assert !IsFeatureImplemented(FEAT_RAS);
            when Fault GPCFOnWalk => result = '100011';
            otherwise => unreachable;
        end;

        return result;
    end;

    case statuscode of
        when Fault AddressSize => result = '0000'::level[1:0]; assert level IN {0,1,2,3};
        when Fault AccessFlag => result = '0010'::level[1:0]; assert level IN {0,1,2,3};
        when Fault Permission => result = '0011'::level[1:0]; assert level IN {0,1,2,3};
        when Fault Translation => result = '0001'::level[1:0]; assert level IN {0,1,2,3};
        when Fault AsyncExternal => result = '010001'; assert UsingAArch32();
        when Fault SyncExternal => result = '010000';
        when Fault SyncExternalOnWalk => result = '0101'::level[1:0]; assert level IN {0,1,2,3};
        when Fault SyncParity => result = '011000';
        when Fault SyncParityOnWalk => result = '0111'::level[1:0]; assert level IN {0,1,2,3};
        when Fault AsyncParity => result = '011001';
        when Fault TagCheck => result = '010001'; assert IsFeatureImplemented(FEAT_MTE2);
        when Fault Alignment => result = '100001';
        when Fault Debug => result = '100010';
        when Fault GPCFOnWalk => result = '1001'::level[1:0]; assert level IN {0,1,2,3};
        when Fault GPCFOnOutput => result = '101000';
        when Fault TLBConflict => result = '110000';
        when Fault UnsupportedAtomicHWUpdate =>
            result = '110001';
        when Fault Lockdown => result = '110100'; // IMPLEMENTATION DEFINED
        when Fault Exclusive => result = '110101'; // IMPLEMENTATION DEFINED
        otherwise => unreachable;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// =====
// Return TRUE if the IPA is reported for the abort

func IPAValid(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault None;

    if fault.gpcf.gpf != GPCF None then
        return fault.secondstage;
    elsif fault.s2fslwalk then
        return fault.statuscode IN {
            Fault AccessFlag,
            Fault Permission,
            Fault Translation,
            Fault AddressSize
        };
    elsif fault.secondstage then
        return fault.statuscode IN {
            Fault AccessFlag,
            Fault Translation,
            Fault AddressSize
        };
    else
        return FALSE;
    end;
end;
```

Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

func IsAsyncAbort(statuscode : Fault) => boolean
begin
    assert statuscode != Fault None;

    return (statuscode IN {Fault AsyncExternal, Fault AsyncParity});
end;

// IsAsyncAbort()
// =====

func IsAsyncAbort(fault : FaultRecord) => boolean
begin
    return IsAsyncAbort(fault.statuscode);
end;
```

Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// =====

func IsDebugException(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault None;
    return fault.statuscode == Fault Debug;
end;
```

Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =====
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

readonly func IsExternalAbort(statuscode : Fault) => boolean
begin
    assert statuscode != Fault\_None;

    return (statuscode IN {
        Fault\_SyncExternal,
        Fault\_SyncParity,
        Fault\_SyncExternalOnWalk,
        Fault\_SyncParityOnWalk,
        Fault\_AsyncExternal,
        Fault\_AsyncParity
    });
end;

// IsExternalAbort()
// =====

readonly func IsExternalAbort(fault : FaultRecord) => boolean
begin
    return IsExternalAbort(fault.statuscode) || fault.gpcf.gpf == GPCF\_EABT;
end;
```

Library pseudocode for shared/functions/aborts/IsExternalAbortOnWalk

```
// IsExternalAbortOnWalk()
// =====

func IsExternalAbortOnWalk(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault\_None;
    return fault.statuscode IN {Fault\_SyncExternalOnWalk, Fault\_SyncParityOnWalk};
end;
```

Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

readonly func IsExternalSyncAbort(statuscode : Fault) => boolean
begin
    assert statuscode != Fault\_None;
    if IsFeatureImplemented(FEAT_RAS) then
        assert ! statuscode IN {Fault\_SyncParity, Fault\_SyncParityOnWalk};
    end;

    return (statuscode IN {
        Fault\_SyncExternal,
        Fault\_SyncParity,
        Fault\_SyncExternalOnWalk,
        Fault\_SyncParityOnWalk
    });
end;

// IsExternalSyncAbort()
// =====

readonly func IsExternalSyncAbort(fault : FaultRecord) => boolean
begin
    return IsExternalSyncAbort(fault.statuscode) || fault.gpcf.gpf == GPCF\_EABT;
end;
```

Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =====
// Return TRUE if a fault is associated with an address descriptor

func IsFault(addrdesc : AddressDescriptor) => boolean
begin
    return addrdesc.fault.statuscode != Fault\_None;
end;

// IsFault()
// =====
// Return TRUE if a fault is associated with a memory access.

func IsFault(fault : Fault) => boolean
begin
    return fault != Fault\_None;
end;

// IsFault()
// =====
// Return TRUE if a fault is associated with status returned by memory.

func IsFault(retstatus : PhysMemRetStatus) => boolean
begin
    return retstatus.statuscode != Fault\_None;
end;
```

Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// =====
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

func IsSErrorInterrupt(statuscode : Fault) => boolean
begin
    assert statuscode != Fault\_None;

    return (statuscode IN {Fault\_AsyncExternal, Fault\_AsyncParity});
end;

// IsSErrorInterrupt()
// =====

func IsSErrorInterrupt(fault : FaultRecord) => boolean
begin
    return IsSErrorInterrupt(fault.statuscode);
end;

// Add a specific type of return value for FaultSyndrome
type IssType of record {
    iss : bits(25),
    iss2 : bits(24)
};
```

Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// =====

func IsSecondStage(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault\_None;

    return fault.secondstage;
end;
```

Library pseudocode for shared/functions/aborts/LInstructionSyndrome

```
// LInstructionSyndrome()
// =====
// Returns the extended syndrome information for a second stage fault.
// [10] - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.
// [9:8] - Access size.
// [7] - Sign extended (for loads).
// [6:2] - Transfer register.
// [1] - Transfer register is 64-bit.
// [0] - Instruction has acquire/release semantics.

impdef func LInstructionSyndrome() => bits(11)
begin
    return ARBITRARY : bits(11);
end;
```

Library pseudocode for shared/functions/aborts/ReportAsGPCEException

```
// ReportAsGPCEException()
// =====
// Determine whether the given GPCF is reported as a Granule Protection Check Exception
// rather than a Data or Instruction Abort

func ReportAsGPCEException(fault : FaultRecord) => boolean
begin
    assert IsFeatureImplemented(FEAT_RME);
    assert fault.statuscode IN {Fault\_GPCFOnWalk, Fault\_GPCFOnOutput};
    assert fault.gpcf.gpf != GPCF\_None;

    if Halted() && EDSCR().SDD == '1' then
        return FALSE;
    end;

    case fault.gpcf.gpf of
        when GPCF\_Walk => return TRUE;
        when GPCF\_AddressSize => return TRUE;
        when GPCF\_EABT => return TRUE;
        when GPCF\_Fail => return SCR_EL3().GPF == '1' && PSTATE.EL != EL3;
    end;
end;
```

Library pseudocode for shared/functions/cache/CACHE_OP

```
// CACHE_OP()
// =====
// Performs Cache maintenance operations as per CacheRecord.

impdef func CACHE_OP(cache : CacheRecord)
begin
    return;
end;
```

Library pseudocode for shared/functions/cache/CPASAtPAS

```
// CPASAtPAS()
// =====
// Get cache PA space for given PA space.

func CPASAtPAS(pas : PASpace) => CachePASpace
begin
    case pas of
        when PAS\_NonSecure =>
            return CPAS\_NonSecure;
        when PAS\_Secure =>
            return CPAS\_Secure;
        when PAS\_Root =>
            return CPAS\_Root;
        when PAS\_Realm =>
            return CPAS\_Realm;
        when PAS\_SystemAgent =>
            return CPAS\_SystemAgent;
        when PAS\_NonSecureProtected =>
            return CPAS\_NonSecureProtected;
        when PAS\_NA6 =>
            return CPAS\_NA6;
        when PAS\_NA7 =>
            return CPAS\_NA7;
        otherwise =>
            unreachable;
    end;
end;
```

Library pseudocode for shared/functions/cache/CPASAtSecurityState

```
// CPASAtSecurityState()
// =====
// Get cache PA space for given security state.

func CPASAtSecurityState(ss : SecurityState) => CachePASpace
begin
  case ss of
    when SS NonSecure =>
      return CPAS NonSecure;
    when SS Secure =>
      return CPAS SecureNonSecure;
    when SS Root =>
      return CPAS Any;
    when SS Realm =>
      return CPAS RealmNonSecure;
  end;
end;
```

Library pseudocode for shared/functions/cache/CacheRecord

```
// CacheRecord
// =====
// Details related to a cache operation.

type CacheRecord of record {
  acctype      : AccessType,           // Access type
  cacheop      : CacheOp,             // Cache operation
  opscope      : CacheOpScope,        // Cache operation type
  cachetype    : CacheType,          // Cache type
  regval       : bits(64),
  paddress     : FullAddress,
  vaddress     : bits(64),             // For VA operations
  setnum       : integer,              // For SW operations
  waynum       : integer,              // For SW operations
  level        : integer,              // For SW operations
  shareability : Shareability,
  is_vmid_valid : boolean,             // is vmid valid for current context
  vmid         : bits(NUM_VMIDBITS),
  is_asid_valid : boolean,             // is asid valid for current context
  asid         : bits(NUM_ASIDBITS),
  security     : SecurityState,
  // For cache operations to full cache or by setnum/waynum
  // For operations by address, PA space in paddress
  cpas : CachePASpace
};
```

Library pseudocode for shared/functions/cache/DecodeSW

```
// DecodeSW()
// =====
// Decode input value into setnum, waynum and level for SW instructions.

func DecodeSW(regval : bits(64), cachetype : CacheType) => (integer, integer, integer)
begin
  let level : integer = UInt(regval[3:1]);
  let (numsets, associativity, linesize) = GetCacheInfo(level, cachetype);
  // For the given level and cachetype, get the number of sets, associativity and
  // cache line size in terms of actual bytes.

  let max_waybits : integer{} = if IsFeatureImplemented(FEAT_CCIDX) then 21 else 10;
  let max_setbits : integer{} = if IsFeatureImplemented(FEAT_CCIDX) then 24 else 15;

  let waybits : integer{} = CeilLog2(associativity) as integer{0..max_waybits};
  let setbits : integer{} = CeilLog2(numsets) as integer{0..max_setbits};
  let linebits : integer{} = FloorLog2(linesize) as integer{4..11};

  let waynum : integer = if associativity == 1 then 0 else UInt(regval[31:32-waybits]);
  let setnum : integer = if numsets == 1 then 0 else UInt(regval[linebits +: setbits]);

  return (setnum, waynum, level);
end;
```

Library pseudocode for shared/functions/cache/GetCacheInfo

```
// GetCacheInfo()
// =====
// Returns numsets, associativity & linesize in terms of actual bytes.

impdef func GetCacheInfo(level : integer, cachetype : CacheType) => (integer, integer, integer)
begin
  let numsets : integer = ImpDefInt("Numsets for DC SW instructions");
  let associativity : integer = ImpDefInt("Associativity for DC SW instructions");
  let linesize : integer = ImpDefInt("Linesize for DC SW instructions");
  return (numsets, associativity, linesize);
end;
```

Library pseudocode for shared/functions/common/IsZeroBit

```
// IsZeroBit()
// =====

func IsZeroBit{N}(x : bits(N)) => bit
begin
  return if IsZero(x) then '1' else '0';
end;
```

Library pseudocode for shared/functions/common/NormalizeReal

```
// NormalizeReal
// =====
// Normalizes x to the form 1.xxx... x 2^y and returns (mantissa, exponent)

func NormalizeReal(x : real) => (real, integer)
begin
  let exponent : integer = ILog2(x);
  let mantissa : real = x / (2.0 ^ exponent);
  return (mantissa, exponent);
end;
```

Library pseudocode for shared/functions/common/RShr

```
// RShr()
// =====
// Shift integer value right with rounding

func RShr(value : integer, shift : integer, round : boolean) => integer
begin
  assert shift > 0;
  if round then
    return (value + (1 << (shift - 1))) >> shift;
  else
    return value >> shift;
  end;
end;
```

Library pseudocode for shared/functions/common/Reverse

```
// Reverse()
// =====
// Reverse subwords of M bits in an N-bit word

func Reverse{N}(word : bits(N), M : integer) => bits(N)
begin
  assert N MOD M == 0;
  var result : bits(N);
  let ssize : integer{} = M as integer{1..N};
  let sw : integer = N DIV ssize;
  for s = 0 to sw-1 do
    result[((sw - 1) - s)*ssize] = word[s*ssize];
  end;
  return result;
end;
```

Library pseudocode for shared/functions/common/Signal

```
// Signal
// =====
// Available signal types

type Signal of enumeration {LOW, HIGH};
```


Library pseudocode for shared/functions/counters

```
var PhysicalCount : bits(88);

var IsLocalTimeoutEventPending : boolean;
var LocalTimeoutVal : bits(64);           // Value to compare against the Virtual Counter Timer
                                           // to generate the local timeout event.
```

Library pseudocode for shared/functions/counters/AArch32_CheckTimerConditions

```
// AArch32_CheckTimerConditions()
// =====
// Checking timer conditions for all A32 timer registers

func AArch32_CheckTimerConditions()
begin
    var status : boolean;
    var offset : bits(64);
    offset = Zeros{64};
    assert !HaveAArch64();
    if HaveEL(EL3) then
        if CNTP_CTL_S().ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_S(),
                                           CNTP_CTL_S().IMASK, InterruptID_CNTPS);
            CNTP_CTL_S().ISTATUS = if status then '1' else '0';
        end;

        if CNTP_CTL_NS().ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_NS(),
                                           CNTP_CTL_NS().IMASK, InterruptID_CNTP);
            CNTP_CTL_NS().ISTATUS = if status then '1' else '0';
        end;
    else
        if CNTP_CTL().ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL(),
                                           CNTP_CTL().IMASK, InterruptID_CNTP);
            CNTP_CTL().ISTATUS = if status then '1' else '0';
        end;
    end;
    if HaveEL(EL2) && CNTHP_CTL().ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTHP_CVAL(),
                                       CNTHP_CTL().IMASK, InterruptID_CNTHP);
        CNTHP_CTL().ISTATUS = if status then '1' else '0';
    end;
    if CNTV_CTL_EL0().ENABLE == '1' then
        status = IsTimerConditionMet(CNTVOFF_EL2(), CNTV_CVAL_EL0(),
                                       CNTV_CTL_EL0().IMASK, InterruptID_CNTV);
        CNTV_CTL_EL0().ISTATUS = if status then '1' else '0';
    end;
    return;
end;
```

Library pseudocode for shared/functions/counters/AArch64_CheckTimerConditions

```
// AArch64_CheckTimerConditions()
// =====
// Checking timer conditions for all A64 timer registers

func AArch64_CheckTimerConditions()
begin
    var status : boolean;
    var offset : bits(64);
    var imask : bit;
    let ss : SecurityState = CurrentSecurityState();
    if (IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && !ELIsInHost(EL0) &&
        CNTHCTL\_EL2().ECV == '1' && SCR\_EL3().ECVEn == '1') then
        offset = CNTPOFF\_EL2();
    else
        offset = Zeros{64};
    end;
    if CNTP\_CTL\_EL0().ENABLE == '1' then
        imask = CNTP\_CTL\_EL0().IMASK;
        if (IsFeatureImplemented(FEAT_RME) && ss IN {SS Root, SS Realm} &&
            CNTHCTL\_EL2().CNTPMASK == '1') then
            imask = '1';
        end;
        status = IsTimerConditionMet(offset, CNTP\_CVAL\_EL0(),
            imask, InterruptID\_CNTP);
        CNTP\_CTL\_EL0().ISTATUS = if status then '1' else '0';
    end;
    if ((HaveEL(EL3) || (HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))) &&
        CNTHP\_CTL\_EL2().ENABLE == '1') then
        status = IsTimerConditionMet(Zeros{64}, CNTHP\_CVAL\_EL2(),
            CNTHP\_CTL\_EL2().IMASK, InterruptID\_CNTHP);
        CNTHP\_CTL\_EL2().ISTATUS = if status then '1' else '0';
    end;
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && CNTHPS\_CTL\_EL2().ENABLE == '1' then
        status = IsTimerConditionMet(Zeros{64}, CNTHPS\_CVAL\_EL2(),
            CNTHPS\_CTL\_EL2().IMASK, InterruptID\_CNTHPS);
        CNTHPS\_CTL\_EL2().ISTATUS = if status then '1' else '0';
    end;

    if CNTPS\_CTL\_EL1().ENABLE == '1' then
        status = IsTimerConditionMet(Zeros{64}, CNTPS\_CVAL\_EL1(),
            CNTPS\_CTL\_EL1().IMASK, InterruptID\_CNTPS);
        CNTPS\_CTL\_EL1().ISTATUS = if status then '1' else '0';
    end;

    if CNTV\_CTL\_EL0().ENABLE == '1' then
        imask = CNTV\_CTL\_EL0().IMASK;
        if (IsFeatureImplemented(FEAT_RME) && ss IN {SS Root, SS Realm} &&
            CNTHCTL\_EL2().CNTVMASK == '1') then
            imask = '1';
        end;
        status = IsTimerConditionMet(CNTVOFF\_EL2(), CNTV\_CVAL\_EL0(),
            imask, InterruptID\_CNTV);
        CNTV\_CTL\_EL0().ISTATUS = if status then '1' else '0';
    end;

    if ((IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || !IsFeatureImplemented(FEAT_SEL2))) &&
        CNTHV\_CTL\_EL2().ENABLE == '1') then
        status = IsTimerConditionMet(Zeros{64}, CNTHV\_CVAL\_EL2(),
            CNTHV\_CTL\_EL2().IMASK, InterruptID\_CNTHV);
        CNTHV\_CTL\_EL2().ISTATUS = if status then '1' else '0';
    end;

    if ((IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_VHE)) &&
        CNTHVS\_CTL\_EL2().ENABLE == '1') then
        status = IsTimerConditionMet(Zeros{64}, CNTHVS\_CVAL\_EL2(),
            CNTHVS\_CTL\_EL2().IMASK, InterruptID\_CNTHVS);
        CNTHVS\_CTL\_EL2().ISTATUS = if status then '1' else '0';
    end;
    return;
end;
```

Library pseudocode for shared/functions/counters/CNTHCTL_EL2_VHE

```
// CNTHCTL_EL2_VHE()
// =====
// In the case where EL2 accesses the CNTHCTL_EL1 register, and the access
// is redirected to CNTHCTL_EL2 as a result of HCR_EL2.E2H being 1,
// then the bits of CNTHCTL_EL2 that are RES0 in CNTHCTL_EL1 are
// treated as being UNKNOWN. This function applies the UNKNOWN behavior.

func CNTHCTL_EL2_VHE(original_value : bits(64)) => bits(64)
begin
    assert PSTATE.EL == EL2;
    assert HCR_EL2().E2H == '1';

    var return_value : bits(64) = original_value;
    if !IsFeatureImplemented(FEAT_NV2p1) then
        return_value[19:18] = ARBITRARY : bits(2);
        return_value[16:10] = ARBITRARY : bits(7);
    end;
    return return_value;
end;
```

Library pseudocode for shared/functions/counters/GenericCounterTick

```
// GenericCounterTick()
// =====
// Increments PhysicalCount value for every clock tick.

func GenericCounterTick()
begin
    var prev_physical_count : bits(64);
    if CNTCR().EN == '0' then
        if !HaveAArch64() then
            AArch32_CheckTimerConditions();
        else
            AArch64_CheckTimerConditions();
        end;
        return;
    end;
    prev_physical_count = PhysicalCountInt();
    if IsFeatureImplemented(FEAT_CNTSC) && CNTCR().SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend{88}(CNTSCR());
    else
        PhysicalCount[87:24] = PhysicalCount[87:24] + 1;
    end;
    if !HaveAArch64() then
        AArch32_CheckTimerConditions();
    else
        AArch64_CheckTimerConditions();
    end;
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;
end;
```

Library pseudocode for shared/functions/counters/IsTimerConditionMet

```
// IsTimerConditionMet()
// =====

func IsTimerConditionMet(offset : bits(64), compare_value : bits(64),
                        imask : bits(1), intid : InterruptID) => boolean
begin
    var condition_met : boolean;
    var level : Signal;
    condition_met = (UInt(PhysicalCountInt() - offset) - UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then HIGH else LOW;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
end;
```

Library pseudocode for shared/functions/counters/SetEventRegister

```
// SetEventRegister()
// =====
// Sets the Event Register of this PE

func SetEventRegister()
begin
    EventRegister = '1';
    return;
end;
```

Library pseudocode for shared/functions/counters/TestEventCNTP

```
// TestEventCNTP()
// =====
// Generate Event stream from the physical counter

func TestEventCNTP(prev_physical_count : bits(64), current_physical_count : bits(64))
begin
    var offset : bits(64);
    var samplebit, previousbit : bit;
    var n : integer;
    if CNTHCTL_EL2().EVNTEN == '1' then
        n = UInt(CNTHCTL_EL2().EVNTI);
        if IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2().EVNTIS == '1' then
            n = n + 8;
        end;
        if (IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled\(\) && !ELIsInHost\(EL0\) &&
            CNTHCTL_EL2().ECV == '1' && SCR_EL3().ECVEN == '1') then
            offset = CNTPOFF_EL2();
        else
            offset = Zeros{64};
        end;
        samplebit = (current_physical_count - offset)[n];
        previousbit = (prev_physical_count - offset)[n];
        if CNTHCTL_EL2().EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister\(\); end;
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister\(\); end;
        end;
    end;
    return;
end;
```

Library pseudocode for shared/functions/counters/TestEventCNTV

```
// TestEventCNTV()
// =====
// Generate Event stream from the virtual counter

func TestEventCNTV(prev_physical_count : bits(64), current_physical_count : bits(64))
begin
    var offset : bits(64);
    var samplebit, previousbit : bit;
    var n : integer;
    if (EffectiveHCR\_EL2\_E2H\(\)::EffectiveTGE\(\) != '11' &&
        CNTKCTL_EL1().EVNTEN == '1') then
        n = UInt(CNTKCTL_EL1().EVNTI);
        if IsFeatureImplemented(FEAT_ECV) && CNTKCTL_EL1().EVNTIS == '1' then
            n = n + 8;
        end;
        offset = if HaveEL\(EL2\) then CNTVOFF_EL2() else Zeros{64};

        samplebit = (current_physical_count - offset)[n];
        previousbit = (prev_physical_count - offset)[n];
        if CNTKCTL_EL1().EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister\(\); end;
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister\(\); end;
        end;
    end;
    return;
end;
```

Library pseudocode for shared/functions/counters/VirtualCounterTimer

```
// VirtualCounterTimer()
// =====
// Returns the Counter-Timer Virtual Count value, the value is as read by CurrentEL to CNTVCT_EL0.

func VirtualCounterTimer() => bits(64)
begin
    var cntvct : bits(64);

    if PSTATE.EL != EL3 then
        if HaveEL(EL2) && !ELIsInHost(PSTATE.EL) then
            cntvct = PhysicalCountInt() - CNTVOFF_EL2();
        else
            cntvct = PhysicalCountInt();
        end;
    else
        if HaveEL(EL2) && !ELUsingAArch32(EL2) then
            cntvct = PhysicalCountInt() - CNTVOFF_EL2();
        elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
            cntvct = PhysicalCountInt() - CNTVOFF();
        else
            cntvct = PhysicalCountInt();
        end;
    end;

    return cntvct;
end;
```

Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// =====

func BitReverse(N)(data : bits(N)) => bits(N)
begin
    var result : bits(N);
    for i = 0 to N-1 do
        result[(N-i)-1] = data[i];
    end;
    return result;
end;
```

Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// =====

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

func Poly32Mod2(N)(data_in : bits(N), poly : bits(32)) => bits(32)
begin
    assert N > 32;
    var data : bits(N) = data_in;
    for i = N-1 downto 32 do
        if data[i] == '1' then
            data[i-1:0] = data[i-1:0] XOR (poly::Zeros(i-32));
        end;
    end;
    return data[31:0];
end;
```

Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
// AESInvMixColumns()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

func AESInvMixColumns(op : bits(128)) => bits(128)
begin
  let in0 : bits(4*8) = op[ 96+:8] :: op[ 64+:8] :: op[ 32+:8] :: op[  0+:8];
  let in1 : bits(4*8) = op[104+:8] :: op[ 72+:8] :: op[ 40+:8] :: op[  8+:8];
  let in2 : bits(4*8) = op[112+:8] :: op[ 80+:8] :: op[ 48+:8] :: op[ 16+:8];
  let in3 : bits(4*8) = op[120+:8] :: op[ 88+:8] :: op[ 56+:8] :: op[ 24+:8];

  var out0 : bits(4*8);
  var out1 : bits(4*8);
  var out2 : bits(4*8);
  var out3 : bits(4*8);

  for c = 0 to 3 do
    out0[c*8+:8] = (FFmul0E(in0[c*8+:8]) XOR FFmul0B(in1[c*8+:8]) XOR FFmul0D(in2[c*8+:8]) XOR
                    FFmul09(in3[c*8+:8]));
    out1[c*8+:8] = (FFmul09(in0[c*8+:8]) XOR FFmul0E(in1[c*8+:8]) XOR FFmul0B(in2[c*8+:8]) XOR
                    FFmul0D(in3[c*8+:8]));
    out2[c*8+:8] = (FFmul0D(in0[c*8+:8]) XOR FFmul09(in1[c*8+:8]) XOR FFmul0E(in2[c*8+:8]) XOR
                    FFmul0B(in3[c*8+:8]));
    out3[c*8+:8] = (FFmul0B(in0[c*8+:8]) XOR FFmul0D(in1[c*8+:8]) XOR FFmul09(in2[c*8+:8]) XOR
                    FFmul0E(in3[c*8+:8]));
  end;

  return (
    out3[3*8+:8] :: out2[3*8+:8] :: out1[3*8+:8] :: out0[3*8+:8] ::
    out3[2*8+:8] :: out2[2*8+:8] :: out1[2*8+:8] :: out0[2*8+:8] ::
    out3[1*8+:8] :: out2[1*8+:8] :: out1[1*8+:8] :: out0[1*8+:8] ::
    out3[0*8+:8] :: out2[0*8+:8] :: out1[0*8+:8] :: out0[0*8+:8]
  );
end;
```

Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()
// =====
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

func AESInvShiftRows(op : bits(128)) => bits(128)
begin
  return (
    op[ 31: 24] :: op[ 55: 48] :: op[ 79: 72] :: op[103: 96] ::
    op[127:120] :: op[ 23: 16] :: op[ 47: 40] :: op[ 71: 64] ::
    op[ 95: 88] :: op[119:112] :: op[ 15:  8] :: op[ 39: 32] ::
    op[ 63: 56] :: op[ 87: 80] :: op[111:104] :: op[  7:  0]
  );
end;
```

Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
// AESInvSubBytes()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

func AESInvSubBytes(op : bits(128)) => bits(128)
begin
  // Inverse S-box values
  let GF2_inv : bits(16*16*8) = (
    /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
    /*F*/ 0x7d0c2155631469e126d677ba7e042b17[127:0] ::
    /*E*/ 0x619953833cbbbec8b0f52aae4d3be0a0[127:0] ::
    /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160[127:0] ::
    /*C*/ 0x5fec8027591012b131c7078833a8dd1f[127:0] ::
    /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc[127:0] ::
    /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147[127:0] ::
    /*9*/ 0x6edf751ce837f9e28535ade72274ac96[127:0] ::
    /*8*/ 0x73e6b4f0cecff297eadc674f4111913a[127:0] ::
    /*7*/ 0x6b8a130103bdafe1020f3fca8f1e2cd0[127:0] ::
    /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890[127:0] ::
    /*5*/ 0x849d8da75746155edab9edfd5048706c[127:0] ::
    /*4*/ 0x92b6655dcc5ca4d41698688664f6f872[127:0] ::
    /*3*/ 0x25d18b6d49a25b76b224d92866a12e08[127:0] ::
    /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54[127:0] ::
    /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c[127:0] ::
    /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952[127:0]
  );
  var out : bits(128);
  for i = 0 to 15 do
    out[i*8+:8] = GF2_inv[UInt(op[i*8+:8])*8+:8];
  end;
  return out;
end;
```

Library pseudocode for shared/functions/crypto/AESMixColumns

```
// AESMixColumns()
// =====
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

func AESMixColumns(op : bits(128)) => bits(128)
begin
    let in0 : bits(4*8) = op[ 96+:8] :: op[ 64+:8] :: op[ 32+:8] :: op[  0+:8];
    let in1 : bits(4*8) = op[104+:8] :: op[ 72+:8] :: op[ 40+:8] :: op[  8+:8];
    let in2 : bits(4*8) = op[112+:8] :: op[ 80+:8] :: op[ 48+:8] :: op[ 16+:8];
    let in3 : bits(4*8) = op[120+:8] :: op[ 88+:8] :: op[ 56+:8] :: op[ 24+:8];

    var out0 : bits(4*8);
    var out1 : bits(4*8);
    var out2 : bits(4*8);
    var out3 : bits(4*8);

    for c = 0 to 3 do
        out0[c*8+:8] = (FFmul02(in0[c*8+:8]) XOR FFmul03(in1[c*8+:8]) XOR
                        in2[c*8+:8] XOR in3[c*8+:8]);
        out1[c*8+:8] = (FFmul02(in1[c*8+:8]) XOR FFmul03(in2[c*8+:8]) XOR
                        in3[c*8+:8] XOR in0[c*8+:8]);
        out2[c*8+:8] = (FFmul02(in2[c*8+:8]) XOR FFmul03(in3[c*8+:8]) XOR
                        in0[c*8+:8] XOR in1[c*8+:8]);
        out3[c*8+:8] = (FFmul02(in3[c*8+:8]) XOR FFmul03(in0[c*8+:8]) XOR
                        in1[c*8+:8] XOR in2[c*8+:8]);
    end;

    return (
        out3[3*8+:8] :: out2[3*8+:8] :: out1[3*8+:8] :: out0[3*8+:8] ::
        out3[2*8+:8] :: out2[2*8+:8] :: out1[2*8+:8] :: out0[2*8+:8] ::
        out3[1*8+:8] :: out2[1*8+:8] :: out1[1*8+:8] :: out0[1*8+:8] ::
        out3[0*8+:8] :: out2[0*8+:8] :: out1[0*8+:8] :: out0[0*8+:8]
    );
end;
```

Library pseudocode for shared/functions/crypto/AESShiftRows

```
// AESShiftRows()
// =====
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

func AESShiftRows(op : bits(128)) => bits(128)
begin
    return (
        op[ 95: 88] :: op[ 55: 48] :: op[ 15:  8] :: op[103: 96] ::
        op[ 63: 56] :: op[ 23: 16] :: op[111:104] :: op[ 71: 64] ::
        op[ 31: 24] :: op[119:112] :: op[ 79: 72] :: op[ 39: 32] ::
        op[127:120] :: op[ 87: 80] :: op[ 47: 40] :: op[  7:  0]
    );
end;
```

Library pseudocode for shared/functions/crypto/AESSubBytes

```
// AESSubBytes()
// =====
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

func AESSubBytes(op : bits(128)) => bits(128)
begin
    // S-box values
    let GF2 : bits(16*16*8) = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c[127:0] ::
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1[127:0] ::
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70[127:0] ::
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba[127:0] ::
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7[127:0] ::
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0[127:0] ::
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160[127:0] ::
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd[127:0] ::
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351[127:0] ::
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0[127:0] ::
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153[127:0] ::
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309[127:0] ::
        /*3*/ 0x75b227ebe28012079a059618c323c704[127:0] ::
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7[127:0] ::
        /*1*/ 0xc072a49cfa2d4adf04759fa7dc982ca[127:0] ::
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63[127:0]
    );
    var out : bits(128);
    for i = 0 to 15 do
        out[i*8+:8] = GF2[UInt(op[i*8+:8])*8+:8];
    end;
    return out;
end;
```

Library pseudocode for shared/functions/crypto/FFmul02

```
// FFmul02()
// =====

func FFmul02(b : bits(8)) => bits(8)
begin
    let FFmul_02 : bits(256*8) = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB[127:0] ::
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB[127:0] ::
        /*D*/ 0xA5A7A1A3ADAF9ABB5B7B1B3BDBFB9BB[127:0] ::
        /*C*/ 0x858781838D8F898B959791939D9F999B[127:0] ::
        /*B*/ 0x656761636D6F696B757771737D7F797B[127:0] ::
        /*A*/ 0x454741434D4F494B555751535D5F595B[127:0] ::
        /*9*/ 0x252721232D2F292B333731333D3F393B[127:0] ::
        /*8*/ 0x050701030D0F090B151711131D1F191B[127:0] ::
        /*7*/ 0xFEFCFAF8F6F4F2F0EECEAE8E6E4E2E0[127:0] ::
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCAC8C6C4C2C0[127:0] ::
        /*5*/ 0xBEBCEB8B8B6B4B2B0AEACAA8A6A4A2A0[127:0] ::
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280[127:0] ::
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260[127:0] ::
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240[127:0] ::
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220[127:0] ::
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200[127:0]
    );
    return FFmul_02[UInt(b)*8+:8];
end;
```


Library pseudocode for shared/functions/crypto/FFmul03

```
// FFmul03()
// =====

func FFmul03(b : bits(8)) => bits(8)
begin
    let FFmul_03 : bits(256*8) = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x1A191C1F16151013020104070E0D080B[127:0] ::
        /*E*/ 0x2A292C2F26252023323134373E3D383B[127:0] ::
        /*D*/ 0x7A797C7F76757073626164676E6D686B[127:0] ::
        /*C*/ 0x4A494C4F46454043525154575E5D585B[127:0] ::
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB[127:0] ::
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB[127:0] ::
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB[127:0] ::
        /*8*/ 0x8A898C8F86858083929194979E9D989B[127:0] ::
        /*7*/ 0x818287848D8E8B88999A9F9C95969390[127:0] ::
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0[127:0] ::
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0[127:0] ::
        /*4*/ 0xD1D2D7D4DDEDEBD8C9CACFCCC5C6C3C0[127:0] ::
        /*3*/ 0x414247444D4E4B48595A5F5C55565350[127:0] ::
        /*2*/ 0x717277747D7E7B78696A6F6C65666360[127:0] ::
        /*1*/ 0x212227242D2E2B28393A3F3C35363330[127:0] ::
        /*0*/ 0x111217141D1E1B18090A0F0C05060300[127:0]
    );
    return FFmul_03[UInt(b)*8+:8];
end;
```

Library pseudocode for shared/functions/crypto/FFmul09

```
// FFmul09()
// =====

func FFmul09(b : bits(8)) => bits(8)
begin
    let FFmul_09 : bits(256*8) = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x464F545D626B70790E071C152A233831[127:0] ::
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1[127:0] ::
        /*D*/ 0x7D746F6659504B42353C272E1118030A[127:0] ::
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A[127:0] ::
        /*B*/ 0x3039222B141D060F78716A635C554E47[127:0] ::
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7[127:0] ::
        /*9*/ 0x0B0219102F263D34434A5158676E757C[127:0] ::
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC[127:0] ::
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD[127:0] ::
        /*6*/ 0x3A3328211E170C05727B6069565F444D[127:0] ::
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6[127:0] ::
        /*4*/ 0x0108131A252C373E49405B526D647F76[127:0] ::
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB[127:0] ::
        /*2*/ 0x4C455E5768617A73040D161F2029323B[127:0] ::
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990[127:0] ::
        /*0*/ 0x777E656C535A41483F362D241B120900[127:0]
    );
    return FFmul_09[UInt(b)*8+:8];
end;
```

Library pseudocode for shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =====

func FFmul0B(b : bits(8)) => bits(8)
begin
    let FFmul_0B : bits(256*8) = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA[127:0] ::
        /*E*/ 0x1318050E3F3429224B405D56676C717A[127:0] ::
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1[127:0] ::
        /*C*/ 0x68637E75444F5259303B262D1C170A01[127:0] ::
        /*B*/ 0x555E434879726F640D061B10212A373C[127:0] ::
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C[127:0] ::
        /*9*/ 0x2E2538330209141F767D606B5A514C47[127:0] ::
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7[127:0] ::
        /*7*/ 0x545F424978736E650C071A11202B363D[127:0] ::
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D[127:0] ::
        /*5*/ 0x2F2439320308151E777C616A5B504D46[127:0] ::
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6[127:0] ::
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB[127:0] ::
        /*2*/ 0x1219040F3E3528234A415C57666D707B[127:0] ::
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0[127:0] ::
        /*0*/ 0x69627F74454E5358313A272C1D160B00[127:0]
    );
    return FFmul_0B[UInt(b)*8+:8];
end;
```

Library pseudocode for shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =====

func FFmul0D(b : bits(8)) => bits(8)
begin
  let FFmul_0D : bits(256*8) = (
    /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
    /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC[127:0] ::
    /*E*/ 0x474A5D50737E69642F2235381B16010C[127:0] ::
    /*D*/ 0x2C21363B1815020F44495E53707D6A67[127:0] ::
    /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7[127:0] ::
    /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1[127:0] ::
    /*A*/ 0x2A27303D1E130409424F5855767B6C61[127:0] ::
    /*9*/ 0x414C5B5675786F622924333E1D10070A[127:0] ::
    /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA[127:0] ::
    /*7*/ 0x4D40575A7974636E25283F32111C0B06[127:0] ::
    /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6[127:0] ::
    /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD[127:0] ::
    /*4*/ 0x262B3C31121F08054E4354597A77606D[127:0] ::
    /*3*/ 0x202D3A3714190E034845525F7C71666B[127:0] ::
    /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB[127:0] ::
    /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADD0D[127:0] ::
    /*0*/ 0x4B46515C7F726568232E3934171A0D00[127:0]
  );
  return FFmul_0D[UInt(b)*8+:8];
end;
```

Library pseudocode for shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =====

func FFmul0E(b : bits(8)) => bits(8)
begin
  let FFmul_0E : bits(256*8) = (
    /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
    /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7[127:0] ::
    /*E*/ 0x6D63717F555B49471D13010F252B3937[127:0] ::
    /*D*/ 0x56584A446E60727C26283A341E10020C[127:0] ::
    /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC[127:0] ::
    /*B*/ 0x202E3C321816040A0505E4C426866747A[127:0] ::
    /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA2886949A[127:0] ::
    /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDFAFA1[127:0] ::
    /*8*/ 0x1B150709232D3F316B657779535D4F41[127:0] ::
    /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896[127:0] ::
    /*6*/ 0x2C22303E141A08065C52404E646A7876[127:0] ::
    /*5*/ 0x17190B052F21333D67697B755F51434D[127:0] ::
    /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD[127:0] ::
    /*3*/ 0x616F7D735957454B111F0D032927353B[127:0] ::
    /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB[127:0] ::
    /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0[127:0] ::
    /*0*/ 0x5A544648626C7E702A243638121C0E00[127:0]
  );
  return FFmul_0E[UInt(b)*8+:8];
end;
```

Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// =====

func SHA256hash(x_in : bits(128), y_in : bits(128), w : bits(128), part1 : boolean) => bits(128)
begin
  var chs, maj, t : bits(32);
  var x : bits(128) = x_in;
  var y : bits(128) = y_in;

  for e = 0 to 3 do
    chs = SHAchoose(y[31:0], y[63:32], y[95:64]);
    maj = SHAmajority{32}(x[31:0], x[63:32], x[95:64]);
    t = y[127:96] + SHA256hashSIGMA1(y[31:0]) + chs + w[e*32];
    x[127:96] = t + x[127:96];
    y[127:96] = t + SHA256hashSIGMA0(x[31:0]) + maj;
    let yx : bits(256) = ROL(y :: x, 32);
    (y, x) = (yx[128+:128], yx[0+:128]);
  end;
  return (if part1 then x else y);
end;
```

Library pseudocode for shared/functions/crypto/SHA256hashSIGMA0

```
// SHA256hashSIGMA0()
// =====

func SHA256hashSIGMA0(x : bits(32)) => bits(32)
begin
    return ROR(x, 2) XOR ROR(x, 13) XOR ROR(x, 22);
end;
```

Library pseudocode for shared/functions/crypto/SHA256hashSIGMA1

```
// SHA256hashSIGMA1()
// =====

func SHA256hashSIGMA1(x : bits(32)) => bits(32)
begin
    return ROR(x, 6) XOR ROR(x, 11) XOR ROR(x, 25);
end;
```

Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// =====

func SHAchoose(x : bits(32), y : bits(32), z : bits(32)) => bits(32)
begin
    return ((y XOR z) AND x) XOR z;
end;
```

Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =====

func SHAmajority{N}(x : bits(N), y : bits(N), z : bits(N)) => bits(N)
begin
    assert N IN {32, 64};
    return ((x AND y) OR ((x OR y) AND z));
end;
```

Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()
// =====

func SHAparity(x : bits(32), y : bits(32), z : bits(32)) => bits(32)
begin
    return (x XOR y XOR z);
end;
```

Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()
// =====
// Used in SM4E crypto instruction

func Sbox(sboxin : bits(8)) => bits(8)
begin
    var sboxout : bits(8);
    let sboxstring : bits(2048) = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xd690e9fecce13db716b614c228fb2c05[127:0] ::
        /*E*/ 0x2b679a762abe04c3aa44132649860699[127:0] ::
        /*D*/ 0x9c4250f491ef987a33540b43edcfac62[127:0] ::
        /*C*/ 0xe4b31ca9c908e89580df94fa758f3fa6[127:0] ::
        /*B*/ 0x4707a7fcf37317ba83593c19e6854fa8[127:0] ::
        /*A*/ 0x686b81b27164da8bf8eb0f4b70569d35[127:0] ::
        /*9*/ 0x1e240e5e6358d1a225227c3b01217887[127:0] ::
        /*8*/ 0xd40046579fd327524c3602e7a0c4c89e[127:0] ::
        /*7*/ 0xeabf8ad240c738b5a3f7f2cef96115a1[127:0] ::
        /*6*/ 0xe0ae5da49b341a55ad933230f58cb1e3[127:0] ::
        /*5*/ 0x1df6e22e8266ca60c02923ab0d534e6f[127:0] ::
        /*4*/ 0xd5db3745defd8e2f03ff6a726d6c5b51[127:0] ::
        /*3*/ 0x8d1baf92bbddbc7f11d95c411f105ad8[127:0] ::
        /*2*/ 0x0ac13188a5cd7bbd2d74d012b8e5b4b0[127:0] ::
        /*1*/ 0x8969974a0c96777e65b9f109c56ec684[127:0] ::
        /*0*/ 0x18f07dec3adc4d2079ee5f3ed7cb3948[127:0]
    );
    let sboxindex : integer = 255 - UInt(sboxin);
    sboxout = sboxstring[sboxindex*8];
    return sboxout;
end;
```

Library pseudocode for shared/functions/decode/DecodeType

```
// DecodeType
// =====

type DecodeType of enumeration {
    Decode_UNDEF,
    Decode_NOP,
    Decode_OK
};
```

Library pseudocode for shared/functions/decode/EndOfDecode

```
// EndOfDecode()
// =====
// This function is invoked to end the Decode phase and performs Branch target Checks
// before taking any UNDEFINED exceptions, NOPs, or continuing to execute.

func EndOfDecode(reason : DecodeType)
begin
    if IsFeatureImplemented(FEAT_BTI) && !UsingAArch32() then
        BranchTargetCheck();
    end;
    case reason of
        when Decode\_NOP => ExecuteAsNOP();
        when Decode\_UNDEF => Undefined();
        when Decode\_OK => pass;           // Continue to execute.
    end;
    return;
end;
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// ClearExclusiveByAddress()
// =====
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.

impdef func ClearExclusiveByAddress(paddress : FullAddress,
                                   processorid : integer,
                                   size : integer)

begin
    return;
end;
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====
// Clear the local Exclusives monitor for the specified processorid.

impdef func ClearExclusiveLocal(processorid : integer)
begin
    return;
end;
```

Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

```
// ExclusiveMonitorsStatus()
// =====
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.

impdef func ExclusiveMonitorsStatus() => bit
begin
    return '0';
end;
```

Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// IsExclusiveGlobal()
// =====
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

impdef func IsExclusiveGlobal(paddress : FullAddress,
                             processorid : integer,
                             size : integer) => boolean
begin
    return TRUE;
end;
```

Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// IsExclusiveLocal()
// =====
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

impdef func IsExclusiveLocal(paddress : FullAddress, processorid : integer,
                             size : integer) => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// MarkExclusiveGlobal()
// =====
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.

impdef func MarkExclusiveGlobal(paddress : FullAddress,
                                processorid : integer,
                                size : integer)
begin
    return;
end;
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// MarkExclusiveLocal()
// =====
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.

impdef func MarkExclusiveLocal(paddress : FullAddress, processorid : integer, size : integer)
begin
    return;
end;
```

Library pseudocode for shared/functions/exclusive/ProcessorID

```
// ProcessorID()
// =====
// Return the ID of the currently executing PE.

impdef func ProcessorID() => integer
begin
    if HaveAArch64() then
        return UInt(MPIDR_EL1().Aff3 :: MPIDR_EL1().Aff2 :: MPIDR_EL1().Aff1 :: MPIDR_EL1().Aff0);
    else
        return UInt(MPIDR().Aff2 :: MPIDR().Aff1 :: MPIDR().Aff0);
    end;
end;
```

Library pseudocode for shared/functions/extension/HaveSoftwareLock

```
// HaveSoftwareLock()
// =====
// Returns TRUE if Software Lock is implemented.

func HaveSoftwareLock(component : Component) => boolean
begin
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        return FALSE;
    end;
    if IsFeatureImplemented(FEAT_DoPD) && component != Component CTI then
        return FALSE;
    end;
    case component of
        when Component ETE =>
            return ImpDefBool("ETE has Software Lock");
        when Component Debug =>
            return ImpDefBool("Debug has Software Lock");
        when Component PMU =>
            return ImpDefBool("PMU has Software Lock");
        when Component CTI =>
            return ImpDefBool("CTI has Software Lock");
        otherwise =>
            unreachable;
    end;
end;
```

Library pseudocode for shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// =====
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

readonly func HaveTraceExt() => boolean
begin
    return IsFeatureImplemented(FEAT_ETE) || IsFeatureImplemented(FEAT_ETMv4);
end;
```

Library pseudocode for shared/functions/extension/InsertIESBBeforeException

```
// InsertIESBBeforeException()
// =====
// Returns an implementation defined choice whether to insert an implicit error synchronization
// barrier before exception.
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.

func InsertIESBBeforeException(el : bits(2)) => boolean
begin
    return (IsFeatureImplemented(FEAT_IESB) && ImpDefBool(
        "Has Implicit Error Synchronization Barrier before Exception"));
end;
```

Library pseudocode for shared/functions/externalaborts/ActionRequired

```
// ActionRequired()
// =====
// Return an implementation specific value:
// returns TRUE if action is required, FALSE otherwise.

impdef func ActionRequired() => boolean
begin
    return TRUE;
end;
```

Library pseudocode for shared/functions/externalaborts/ClearPendingDelegatedSError

```
// ClearPendingDelegatedSError()
// =====
// Clear a pending delegated SError interrupt.

func ClearPendingDelegatedSError()
begin
    assert IsFeatureImplemented(FEAT_E3DSE);
    SCR_EL3().DSE = '0';
end;
```

Library pseudocode for shared/functions/externalaborts/ClearPendingPhysicalSError

```
// ClearPendingPhysicalSError()
// =====
// Clear a pending physical SError interrupt.

impdef func ClearPendingPhysicalSError()
begin
    return;
end;
```

Library pseudocode for shared/functions/externalaborts/ClearPendingVirtualSError

```
// ClearPendingVirtualSError()
// =====
// Clear a pending virtual SError interrupt.

func ClearPendingVirtualSError()
begin
    if ELUsingAArch32\(EL2\) then
        HCR().VA = '0';
    else
        HCR_EL2().VSE = '0';
    end;
end;
```

Library pseudocode for shared/functions/externalaborts/ErrorIsContained

```
// ErrorIsContained()
// =====
// Return an implementation specific value:
// TRUE if Error is contained by the PE, FALSE otherwise.

impdef func ErrorIsContained() => boolean
begin
    return TRUE;
end;
```

Library pseudocode for shared/functions/externalaborts/ErrorIsSynchronized

```
// ErrorIsSynchronized()
// =====
// Return an implementation specific value:
// returns TRUE if Error is synchronized by any synchronization event
// FALSE otherwise.

impdef func ErrorIsSynchronized() => boolean
begin
    return TRUE;
end;
```

Library pseudocode for shared/functions/externalaborts/ExtAbortToAArch64

```
// ExtAbortToAArch64()
// =====
// Returns TRUE if synchronous exception is being taken to an Exception level using AArch64_

func ExtAbortToAArch64(fault : FaultRecord) => boolean
begin
    assert IsExternalSyncAbort(fault.statuscode);

    return !ELUsingAArch32\(SyncExternalAbortTarget\)(fault);
end;
```

Library pseudocode for shared/functions/externalaborts/ExternalAbort

```
// ExternalAbort()
// =====

func ExternalAbort(fault : FaultRecord) recurselimit 1
begin
    if IsExternalSyncAbort(fault) then
        if UsingAArch32\(\) then
            AArch32\_Abort(fault);
        else
            AArch64\_Abort(fault);
        end;
    else
        PendSErrorInterrupt(fault);
    end;
end;
```

Library pseudocode for shared/functions/externalaborts/ExternalFault

```
// ExternalFault()
// =====
// Return a fault recording indicating a fault for a Synchronous/Asynchronous External abort.

func ExternalFault(memretstatus : PhysMemRetStatus, iswrite : boolean,
                  memaddrdesc : AddressDescriptor, size : integer,
                  accdesc : AccessDescriptor) => FaultRecord
begin
    assert (memretstatus.statuscode IN {Fault\_SyncExternal, Fault\_AsyncExternal} ||
            (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode IN {Fault\_SyncParity,
                                                                           Fault\_AsyncParity}));

    var fault : FaultRecord = NoFault(accdesc, memaddrdesc.vaddress);
    fault.statuscode = memretstatus.statuscode;
    fault.write      = iswrite;
    fault.extflag    = memretstatus.extflag;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
        ((IsFeatureImplemented(FEAT_RASv2) && ExtAbortToAArch64(fault) &&
          PEErrorState(fault) IN {ErrorState\_UC, ErrorState\_UEU})) ||
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                           size, accdesc))) then
        if fault.statuscode == Fault\_SyncParity then
            fault.statuscode = Fault\_AsyncParity;
        else
            fault.statuscode = Fault\_AsyncExternal;
        end;
    end;

    if IsFeatureImplemented(FEAT_RAS) then
        fault.merrorstate = memretstatus.merrorstate;
    end;
    fault.paddress = memaddrdesc.paddress;
    return fault;
end;
```

Library pseudocode for shared/functions/externalaborts/FaultIsCorrected

```
// FaultIsCorrected()
// =====
// Return an implementation specific value:
// TRUE if fault is corrected by the PE, FALSE otherwise.

readonly impdef func FaultIsCorrected() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/externalaborts/GetPendingPhysicalSError

```
// GetPendingPhysicalSError()
// =====
// Returns the FaultRecord containing details of pending Physical SError
// interrupt.

impdef func GetPendingPhysicalSError() => FaultRecord
begin
    return ARBITRARY : FaultRecord;
end;
```

Library pseudocode for shared/functions/externalaborts/HandleExternalAbort

```
// HandleExternalAbort()
// =====
// Takes a Synchronous/Asynchronous abort based on fault.

func HandleExternalAbort(memretstatus : PhysMemRetStatus, iswrite : boolean,
                        memaddrdesc : AddressDescriptor, size : integer,
                        accdesc : AccessDescriptor)
begin
    assert (memretstatus.statuscode IN {Fault\_SyncExternal, Fault\_AsyncExternal} ||
            (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode IN {Fault\_SyncParity,
                                                                           Fault\_AsyncParity}));

    let fault : FaultRecord = ExternalFault(memretstatus, iswrite, memaddrdesc, size, accdesc);
    ExternalAbort(fault);
end;
```


Library pseudocode for shared/functions/externalaborts/HandleExternalReadAbort

```
// HandleExternalReadAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

func HandleExternalReadAbort(memstatus : PhysMemRetStatus, memaddrdesc : AddressDescriptor,
                             size : integer, accdesc : AccessDescriptor)
begin
    let iswrite : boolean = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
end;
```

Library pseudocode for shared/functions/externalaborts/HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// =====
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

func HandleExternalTTWAbort(memretstatus : PhysMemRetStatus, iswrite : boolean,
                             memaddrdesc : AddressDescriptor,
                             accdesc : AccessDescriptor, size : integer,
                             input_fault : FaultRecord) => FaultRecord
begin
    var output_fault : FaultRecord = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
        ((IsFeatureImplemented(FEAT_RASv2) && ExtAbortToAArch64(output_fault) &&
          PEErrorState(output_fault) IN {ErrorState\_UC, ErrorState\_UEU})) ||
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                           size, accdesc)) then
        if output_fault.statuscode == Fault\_SyncParity then
            output_fault.statuscode = Fault\_AsyncParity;
        else
            output_fault.statuscode = Fault\_AsyncExternal;
        end;
    end;

    // If a synchronous fault is on a translation table walk, then update the fault type.
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault\_SyncParity then
            output_fault.statuscode = Fault\_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault\_SyncExternalOnWalk;
        end;
    end;
    if IsFeatureImplemented(FEAT_RAS) then
        output_fault.merrorstate = memretstatus.merrorstate;
    end;
    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault\_None;
    end;
    output_fault.paddress = memaddrdesc.paddress;
    return output_fault;
end;
```

Library pseudocode for shared/functions/externalaborts/HandleExternalWriteAbort

```
// HandleExternalWriteAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

func HandleExternalWriteAbort(memstatus : PhysMemRetStatus, memaddrdesc : AddressDescriptor,
                              size : integer, accdesc : AccessDescriptor)
begin
    let iswrite : boolean = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
end;
```

Library pseudocode for shared/functions/externalaborts/IsDelegatedSErrorPending

```
// IsDelegatedSErrorPending()
// =====
// Return TRUE if a delegated SError interrupt is pending.

func IsDelegatedSErrorPending() => boolean
begin
    return SCR_EL3().DSE == '1';
end;
```

Library pseudocode for shared/functions/externalaborts/IsExternalAbortTakenSynchronously

```
// IsExternalAbortTakenSynchronously()
// =====
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.

impdef func IsExternalAbortTakenSynchronously(memstatus : PhysMemRetStatus, iswrite : boolean,
                                             desc : AddressDescriptor, size : integer,
                                             accdesc : AccessDescriptor) => boolean

begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/externalaborts/IsPhysicalSErrorPending

```
// IsPhysicalSErrorPending()
// =====
// Returns TRUE if a physical SError interrupt is pending.

impdef func IsPhysicalSErrorPending() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/externalaborts/IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =====
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

func IsSErrorEdgeTriggered() => boolean
begin
    if IsFeatureImplemented(FEAT_DoubleFault) then
        return TRUE;
    else
        return ImpDefBool("Edge-triggered SError");
    end;
end;
```

Library pseudocode for shared/functions/externalaborts/IsSynchronizablePhysicalSErrorPending

```
// IsSynchronizablePhysicalSErrorPending()
// =====
// Returns TRUE if a synchronizable physical SError interrupt is pending.

impdef func IsSynchronizablePhysicalSErrorPending() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/externalaborts/IsVirtualSErrorPending

```
// IsVirtualSErrorPending()
// =====
// Return TRUE if a virtual SError interrupt is pending.

func IsVirtualSErrorPending() => boolean
begin
    if ELUsingAArch32\(EL2\) then
        return HCR().VA == '1';
    else
        return HCR_EL2().VSE == '1';
    end;
end;
```

Library pseudocode for shared/functions/externalaborts/PEErrorState

```
// PEErrorState()
// =====
// Returns the error state of the PE on taking an error exception:
// The PE error state reported to software through the exception syndrome also
// depends on how the exception is taken, and so might differ from the value
// returned from this function.

func PEErrorState(fault : FaultRecord) => ErrorState
begin
    assert !FaultIsCorrected();
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
        ReportErrorAsUC()) then
        return ErrorState UC;
    end;

    if !StateIsRecoverable() || ReportErrorAsUEU() then
        return ErrorState UEU;
    end;

    if ActionRequired() || ReportErrorAsUER() then
        return ErrorState UER;
    end;

    return ErrorState UEO;
end;
```

Library pseudocode for shared/functions/externalaborts/PendErrorInterrupt

```
// PendErrorInterrupt()
// =====
// Pend the SError Interrupt.

impdef func PendErrorInterrupt(fault : FaultRecord)
begin
    return;
end;
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsIMPDEF

```
// ReportErrorAsIMPDEF()
// =====
// Return an implementation specific value:
// returns TRUE if Error is IMPDEF, FALSE otherwise.

impdef func ReportErrorAsIMPDEF() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsUC

```
// ReportErrorAsUC()
// =====
// Return an implementation specific value:
// returns TRUE if Error is Uncontainable, FALSE otherwise.

impdef func ReportErrorAsUC() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsUER

```
// ReportErrorAsUER()
// =====
// Return an implementation specific value:
// returns TRUE if Error is Recoverable, FALSE otherwise.

impdef func ReportErrorAsUER() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsUEU

```
// ReportErrorAsUEU()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is Unrecoverable, FALSE otherwise.  
  
impdef func ReportErrorAsUEU() => boolean  
begin  
    return FALSE;  
end;
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsUncategorized

```
// ReportErrorAsUncategorized()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is uncategorized, FALSE otherwise.  
  
impdef func ReportErrorAsUncategorized() => boolean  
begin  
    return FALSE;  
end;
```

Library pseudocode for shared/functions/externalaborts/StateIsRecoverable

```
// StateIsRecoverable()  
// =====  
// Return an implementation specific value:  
// returns TRUE is PE State is unrecoverable else FALSE.  
  
impdef func StateIsRecoverable() => boolean  
begin  
    return TRUE;  
end;
```

Library pseudocode for shared/functions/float/bfloat/BFAdd

```
// BFAdd()
// =====
// Non-widening BFloat16 addition used by SVE2 instructions.

func BFAdd{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  let fpexc : boolean = TRUE;
  return BFAdd{N}(op1, op2, fpcr, fpexc);
end;

// BFAdd()
// =====
// Non-widening BFloat16 addition following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 + op2.
// The 'fpcr' argument supplies the FPCR control bits.

func BFAdd{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
  assert N == 16;
  let rounding : FPRounding = FPRoundingMode(fpcr);
  var done : boolean;
  var result : bits(2*N);

  let op1_s : bits(2*N) = op1 :: Zeros{N};
  let op2_s : bits(2*N) = op2 :: Zeros{N};
  let (type1,sign1,value1) = FPUnpack{2*N}(op1_s, fpcr, fpexc);
  let (type2,sign2,value2) = FPUnpack{2*N}(op2_s, fpcr, fpexc);

  (done,result) = FPProcessNaNs{2*N}(type1, type2, op1_s, op2_s, fpcr, fpexc);

  if !done then
    let inf1 = (type1 == FPType_Infinity);
    let inf2 = (type2 == FPType_Infinity);
    let zero1 = (type1 == FPType_Zero);
    let zero2 = (type2 == FPType_Zero);

    if inf1 && inf2 && sign1 == NOT(sign2) then
      result = FPDefaultNaN{2*N}(fpcr);
      if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
      result = FPInfinity{2*N}('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
      result = FPInfinity{2*N}('1');
    elsif zero1 && zero2 && sign1 == sign2 then
      result = FPZero{2*N}(sign1);
    else
      let result_value : real = value1 + value2;
      if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero{2*N}(result_sign);
      else
        result = FPRoundBF{2*N}(result_value, fpcr, rounding, fpexc);
      end;
    end;

    if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr); end;
  end;

  return result[2*N-1:N];
end;
```

Library pseudocode for shared/functions/float/bfloat/BFAdd_ZA

```
// BFAdd_ZA()
// =====
// Non-widening BFloat16 addition used by SME2 ZA-targeting instructions.

func BFAdd_ZA(op1 : bits(16), op2 : bits(16), fpcr_in : FPCR_Type) => bits(16)
begin
  let fpexc : boolean = FALSE;
  var fpcr : FPCR_Type = fpcr_in;
  fpcr.DN = '1'; // Generate default NaN values
  return BFAdd{16}(op1, op2, fpcr, fpexc);
end;
```

Library pseudocode for shared/functions/float/bfloat/BFClampScale

```
// BFClampScale()
// =====

func BFClampScale(op : bits(16), scale_in : integer) => integer
begin
  let E : integer = 8;
  let F : integer = 7;
  let exp : integer = UInt(op[14:7]);
  let emax : integer = (1 << E) - 1;
  let min_scale : integer = -(F + 1);
  let max_scale : integer = emax + (F + 1);

  let scale : integer = Max(min_scale - exp, Min(scale_in, max_scale - exp));
  return scale;
end;
```

Library pseudocode for shared/functions/float/bfloat/BFDotAdd

```
// BFDotAdd()
// =====
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

func BFDotAdd(addend : bits(32), op1_a : bits(16), op1_b : bits(16),
              op2_a : bits(16), op2_b : bits(16), fpcr_in : FPCR_Type) => bits(32)
begin
  var fpcr : FPCR_Type = fpcr_in;
  var prod : bits(32);

  var result : bits(32);
  if !IsFeatureImplemented(FEAT_EBF16) || fpcr.EBF == '0' then // Standard BFloat16 behaviors
    prod = FPAdd\_BF16(BFMulH(op1_a, op2_a, fpcr), BFMulH(op1_b, op2_b, fpcr), fpcr);
    result = FPAdd\_BF16(addend, prod, fpcr);
  else // Extended BFloat16 behaviors
    let isbfloat16 : boolean = TRUE;
    let fpexc : boolean = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1'; // Generate default NaN values
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    result = FPAdd{32}(addend, prod, fpcr, fpexc);
  end;

  return result;
end;
```

Library pseudocode for shared/functions/float/bfloat/BFInfinity

```
// BFInfinity()
// =====

func BFInfinity{N}(sign : bit) => bits(N)
begin
  assert N == 16;
  let E : integer{} = 8;
  let F : integer{} = N - (E + 1);
  return sign :: Ones{E} :: Zeros{F};
end;
```

Library pseudocode for shared/functions/float/bfloat/BFMatMulAddH

```
// BFMatMulAddH()
// =====
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

func BFMatMulAddH(addend : bits(128), op1 : bits(128),
                  op2 : bits(128), fpcr : FPCR_Type) => bits(128)
begin
    var result : bits(128);
    var sum : bits(32);

    for i = 0 to 1 do
        for j = 0 to 1 do
            sum = addend[(2*i + j)*:32];
            for k = 0 to 1 do
                let elt1_a : bits(16) = op1[(4*i + 2*k + 0)*:16];
                let elt1_b : bits(16) = op1[(4*i + 2*k + 1)*:16];
                let elt2_a : bits(16) = op2[(4*j + 2*k + 0)*:16];
                let elt2_b : bits(16) = op2[(4*j + 2*k + 1)*:16];
                sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, fpcr);
            end;
            result[(2*i + j)*:32] = sum;
        end;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/bfloat/BFMax

```
// BFMax()
// =====
// BFloat16 maximum.

func BFMax{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
  let fpecx : boolean = TRUE;
  return BFMax{N}(op1, op2, fpcr, altfp, fpecx);
end;

// BFMax()
// =====
// BFloat16 maximum.

func BFMax{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, altfp : boolean) => bits(N)
begin
  let fpecx : boolean = TRUE;
  return BFMax{N}(op1, op2, fpcr, altfp, fpecx);
end;

// BFMax()
// =====
// BFloat16 maximum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the larger value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

func BFMax{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type,
  altfp : boolean, fpecx : boolean) => bits(N)
begin
  assert N == 16;
  var fpcr : FPCR_Type = fpcr_in;
  let rounding : FPRounding = FPRoundingMode(fpcr);
  var done : boolean;
  var result : bits(2*N);

  let op1_s : bits(2*N) = op1 :: Zeros{N};
  let op2_s : bits(2*N) = op2 :: Zeros{N};
  let (type1,sign1,value1) = FPUnpack{2*N}(op1_s, fpcr, fpecx);
  let (type2,sign2,value2) = FPUnpack{2*N}(op2_s, fpcr, fpecx);

  if altfp && type1 == FType_Zero && type2 == FType_Zero && sign1 != sign2 then
    // Alternate handling of zeros with differing sign
    return BFZero{N}(sign2);
  elseif altfp && (type1 IN {FType_SNaN, FType_QNaN}) || type2 IN {FType_SNaN, FType_QNaN} then
    // Alternate handling of NaN inputs
    if fpecx then FProcessException(FPExc_InvalidOp, fpcr); end;
    return (if type2 == FType_Zero then BFZero{N}(sign2) else op2);
  end;

  (done,result) = FProcessNaNs{2*N}(type1, type2, op1_s, op2_s, fpcr, fpecx);
  if !done then
    var fptype : FType;
    var sign : bit;
    var value : real;
    if value1 > value2 then
      (fptype,sign,value) = (type1,sign1,value1);
    else
      (fptype,sign,value) = (type2,sign2,value2);
    end;
    if fptype == FType_Infinity then
      result = FPinfinity{2*N}(sign);
    elseif fptype == FType_Zero then
      sign = sign1 AND sign2; // Use most positive sign
      result = FPZero{2*N}(sign);
    else
      if altfp then // Denormal output is not flushed to zero
        fpcr.FZ = '0';
      end;
      result = FPRoundBF{2*N}(value, fpcr, rounding, fpecx);
    end;

    if fpecx then FProcessDenorms(type1, type2, 2*N, fpcr); end;
  end;

  return result[2*N-1:N];
end;
```


Library pseudocode for shared/functions/float/bfloat/BFMaxNum

```
// BFMaxNum()
// =====

func BFMaxNum{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return BFMaxNum{N}(op1, op2, fpcr, fpexc);
end;

// BFMaxNum()
// =====
// BFloat16 maximum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the larger number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

func BFMaxNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N == 16;
    let isbfloat16 : boolean = TRUE;
    var op1 : bits(N) = op1_in;
    var op2 : bits(N) = op2_in;
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    var result : bits(N);

    let (type1,-,-) = FPUnpackBase{N}(op1, fpcr, fpexc, isbfloat16);
    let (type2,-,-) = FPUnpackBase{N}(op2, fpcr, fpexc, isbfloat16);

    let type1_nan : boolean = type1 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    let type2_nan : boolean = type2 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPTYPE\_QNaN && type2 != FPTYPE\_QNaN then
            op1 = BFInfinity{N}('1');
        elseif type1 != FPTYPE\_QNaN && type2 == FPTYPE\_QNaN then
            op2 = BFInfinity{N}('1');
        end;
    end;

    let altfmaxfmin : boolean = FALSE; // Do not use alternate NaN handling
    result = BFMax{N}(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
end;
```

Library pseudocode for shared/functions/float/bfloat/BFMin

```
// BFMin()
// =====
// BFloat16 minimum.

func BFMin{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = TRUE;
    return BFMin{N}(op1, op2, fpcr, altfp, fpexc);
end;

// BFMin()
// =====
// BFloat16 minimum.

func BFMin{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, altfp : boolean) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return BFMin{N}(op1, op2, fpcr, altfp, fpexc);
end;

// BFMin()
// =====
// BFloat16 minimum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

func BFMin{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type,
    altfp : boolean, fpexc : boolean) => bits(N)
begin
    assert N == 16;
    var fpcr : FPCR_Type = fpcr_in;
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(2*N);

    let op1_s : bits(2*N) = op1 :: Zeros{N};
    let op2_s : bits(2*N) = op2 :: Zeros{N};
    let (type1,sign1,value1) = FPUnpack{2*N}(op1_s, fpcr, fpexc);
    let (type2,sign2,value2) = FPUnpack{2*N}(op2_s, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero{N}(sign2);
    elseif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        return (if type2 == FPType_Zero then BFZero{N}(sign2) else op2);
    end;

    (done,result) = FPProcessNaNs{2*N}(type1, type2, op1_s, op2_s, fpcr, fpexc);
    if !done then
        var fptype : FPType;
        var sign : bit;
        var value : real;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        end;
        if fptype == FPType_Infinity then
            result = FPInfinity{2*N}(sign);
        elseif fptype == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero{2*N}(sign);
        else
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            end;
            result = FPRoundBF{2*N}(value, fpcr, rounding, fpexc);
        end;

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr); end;
    end;

    return result[2*N-1:N];
end;
```

Library pseudocode for shared/functions/float/bfloat/BFMinNum

```
// BFMinNum()
// =====

func BFMinNum{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return BFMinNum{N}(op1, op2, fpcr, fpexc);
end;

// BFMinNum()
// =====
// BFloat16 minimum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

func BFMinNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N == 16;
    let isbfloat16 : boolean = TRUE;
    var op1 : bits(N) = op1_in;
    var op2 : bits(N) = op2_in;
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    var result : bits(N);

    let (type1,-,-) = FPUnpackBase{N}(op1, fpcr, fpexc, isbfloat16);
    let (type2,-,-) = FPUnpackBase{N}(op2, fpcr, fpexc, isbfloat16);

    let type1_nan : boolean = type1 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    let type2_nan : boolean = type2 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPTYPE\_QNaN && type2 != FPTYPE\_QNaN then
            op1 = BFInfinity{N}('0');
        elseif type1 != FPTYPE\_QNaN && type2 == FPTYPE\_QNaN then
            op2 = BFInfinity{N}('0');
        end;
    end;

    let altfmaxfmin : boolean = FALSE; // Do not use alternate NaN handling
    result = BFMin{N}(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
end;
```

Library pseudocode for shared/functions/float/bfloat/BFMul

```
// BFMul()
// =====
// Non-widening BFloat16 multiply used by SVE2 instructions.

func BFMul(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = TRUE;
    return BFMul(op1, op2, fpcr, fpexc);
end;

// BFMul()
// =====
// Non-widening BFloat16 multiply following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 * op2.
// The 'fpcr' argument supplies the FPCR control bits.

func BFMul(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type, fpexc : boolean) => bits(16)
begin
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(32);

    let op1_s : bits(32) = op1 :: Zeros{16};
    let op2_s : bits(32) = op2 :: Zeros{16};
    let (type1, sign1, value1) = FPUnpack{32}(op1_s, fpcr, fpexc);
    let (type2, sign2, value2) = FPUnpack{32}(op2_s, fpcr, fpexc);

    (done, result) = FPProcessNaNs{32}(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        let inf1 = (type1 == FPType Infinity);
        let inf2 = (type2 == FPType Infinity);
        let zero1 = (type1 == FPType Zero);
        let zero2 = (type2 == FPType Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc InvalidOp, fpcr); end;
        elsif inf1 || inf2 then
            result = FPInfinity{32}(sign1 XOR sign2);
        elsif zero1 || zero2 then
            result = FPZero{32}(sign1 XOR sign2);
        else
            result = FPRoundBF{32}(value1*value2, fpcr, rounding, fpexc);
        end;

        if fpexc then FPProcessDenorms(type1, type2, 32, fpcr); end;
    end;

    return result[31:16];
end;
```



```

// BFMulAdd()
// =====
// Non-widening BFloat16 fused multiply-add used by SVE2 instructions.

func BFMulAdd(addend : bits(16), op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = TRUE;
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);
end;

// BFMulAdd()
// =====
// Non-widening BFloat16 fused multiply-add following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates addend + op1*op2 with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits.

func BFMulAdd(addend : bits(16), op1 : bits(16), op2 : bits(16),
              fpcr : FPCR_Type, fpexc : boolean) => bits(16)
begin
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(32);

    let addend_s : bits(32) = addend :: Zeros{16};
    let op1_s : bits(32) = op1 :: Zeros{16};
    let op2_s : bits(32) = op2 :: Zeros{16};
    let (typeA, signA, valueA) = FPUnpack{32}(addend_s, fpcr, fpexc);
    let (type1, sign1, value1) = FPUnpack{32}(op1_s, fpcr, fpexc);
    let (type2, sign2, value2) = FPUnpack{32}(op2_s, fpcr, fpexc);

    let inf1 = (type1 == FPType Infinity);
    let inf2 = (type2 == FPType Infinity);
    let zero1 = (type1 == FPType Zero);
    let zero2 = (type2 == FPType Zero);

    (done, result) = FPProcessNaNs3{32}(typeA, type1, type2, addend_s, op1_s, op2_s, fpcr, fpexc);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc InvalidOp, fpcr); end;
        end;
    end;

    if !done then
        let infA = (typeA == FPType Infinity);
        let zeroA = (typeA == FPType Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        let signP = sign1 XOR sign2;
        let infP = inf1 || inf2;
        let zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        let invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc InvalidOp, fpcr); end;

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity{32}('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity{32}('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero{32}(signA);

        // Otherwise calculate numerical result and round it.
        else
            let result_value : real = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding NEGINF then '1' else '0';
                result = FPZero{32}(result_sign);
            else
                result = FPRoundBF{32}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if !invalidop && fpexc then
            FPProcessDenorms3(typeA, type1, type2, 32, fpcr);
        end;
    end;
end;

```

```

    return result[31:16];
end;

```

Library pseudocode for shared/functions/float/bfloat/BFMulAddH

```

// BFMulAddH()
// =====
// Used by BFMLALB, BFMLALT, BFMLSBLB and BFMLSILT instructions.

func BFMulAddH(addend : bits(32), op1 : bits(16), op2 : bits(16), fpcr_in : FPCR_Type) => bits(32)
begin
    let value1 : bits(32) = op1 :: Zeros{16};
    let value2 : bits(32) = op2 :: Zeros{16};
    var fpcr : FPCR_Type = fpcr_in;
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    // When using alternative floating-point behavior, do not generate floating-point exceptions
    let fpexc : boolean = !altfp;
    if altfp then fpcr.[FIZ,FZ] = '11'; end;
    // Flush denormal input and
    // output to zero
    if altfp then fpcr.RMode = '00'; end;
    // Use RNE rounding mode
    return FPMulAdd{32}(addend, value1, value2, fpcr, fpexc);
end;

```

Library pseudocode for shared/functions/float/bfloat/BFMulAddH_ZA

```

// BFMulAddH_ZA()
// =====
// Used by SME2 ZA-targeting BFMLAL and BFMLSIL instructions.

func BFMulAddH_ZA(addend : bits(32), op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(32)
begin
    let value1 : bits(32) = op1 :: Zeros{16};
    let value2 : bits(32) = op2 :: Zeros{16};
    return FPMulAdd\_ZA{32}(addend, value1, value2, fpcr);
end;

```

Library pseudocode for shared/functions/float/bfloat/BFMulAdd_ZA

```

// BFMulAdd_ZA()
// =====
// Non-widening BFloat16 fused multiply-add used by SME2 ZA-targeting instructions.

func BFMulAdd_ZA(addend : bits(16), op1 : bits(16), op2 : bits(16), fpcr_in : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = FALSE;
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);
end;

```

Library pseudocode for shared/functions/float/bfloat/BFMulH

```

// BFMulH()
// =====
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

func BFMulH(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(32)
begin
    var result : bits(32);

    let (type1,sign1,value1) = BFUnpack{16}(op1);
    let (type2,sign2,value2) = BFUnpack{16}(op2);
    if type1 == FPTYPE\_QNaN || type2 == FPTYPE\_QNaN then
        result = FPDefaultNaN{32}(fpcr);
    else
        let inf1 = (type1 == FPTYPE\_Infinity);
        let inf2 = (type2 == FPTYPE\_Infinity);
        let zero1 = (type1 == FPTYPE\_Zero);
        let zero2 = (type2 == FPTYPE\_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN{32}(fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity{32}(sign1 XOR sign2);
        elsif zero1 || zero2 then
            result = FPZero{32}(sign1 XOR sign2);
        else
            result = BFRound(value1*value2);
        end;
    end;

    return result;
end;

```

Library pseudocode for shared/functions/float/bfloat/BFNeg

```
// BFNeg()
// =====

func BFNeg(op : bits(16)) => bits(16)
begin
    let honor_alftp : boolean = TRUE;    // Honor alternate handling
    return BFNeg(op, honor_alftp);
end;

// BFNeg()
// =====

func BFNeg(op : bits(16), honor_alftp : boolean) => bits(16)
begin
    if honor_alftp && !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        if FPCR().AH == '1' then
            let fpexc : boolean = FALSE;
            let isbfloat16 : boolean = TRUE;
            let (fptype, -, -) = FPUnpackBase{16}(op, FPCR(), fpexc, isbfloat16);
            if fptype IN {FPTType\_SNaN, FPType\_QNaN} then
                return op;    // When FPCR().AH=1, sign of NaN has no consequence
            end;
        end;
    end;
    return NOT(op[15]) :: op[14:0];
end;
```

Library pseudocode for shared/functions/float/bfloat/BFRound

```
// BFRound()
// =====
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

func BFRound(op : real) => bits(32)
begin
    assert op != 0.0;
    var result : bits(32);

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    let minimum_exp : integer = -126; let E : integer{} = 8; let F : integer{} = 23;

    // Split value into sign, unrounded mantissa and exponent.
    var sign : bit;
    var exponent : integer;
    var mantissa : real;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    end;

    (mantissa, exponent) = NormalizeReal(mantissa);
    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero{32}(sign);
    end;

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    let biased_exp : integer = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent); end;

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    var int_mant : integer = RoundDown(mantissa * 2.0^F);
    let error : real = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 && int_mant[0] == '0' then
        int_mant = int_mant + 1;
    end;

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity{32}(sign);    // Overflows generate appropriately-signed Infinity
    else
        result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0];
    end;

    return result;
end;
```


Library pseudocode for shared/functions/float/bfloat/BFScale

```
// BFScale()
// =====
// Scales BFloat16 operand by 2.0 to the power of the signed integer value.

func BFScale(op : bits(16), scale : integer, fpcr : FPCR_Type) => bits(16)
begin
    var result : bits(32);

    let op_s : bits(32) = op :: Zeros{16};
    let (fptype,sign,value) = FPUnpack{32}(op_s, fpcr);

    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        result = FPProcessNaN{32}(fptype, op_s, fpcr);
    elsif fptype == FPTYPE\_Zero then
        result = FPZero{32}(sign);
    elsif fptype == FPTYPE\_Infinity then
        result = FPIInfinity{32}(sign);
    else
        let rounding : FPRounding = FPRoundingMode(fpcr);
        let fpexc : boolean = TRUE;
        let clamped_scale : integer = BFClampScale(op, scale);
        result = FPRoundBF{32}(value * (2.0^clamped_scale), fpcr, rounding, fpexc);
        if fpexc then FPProcessDenorm(fptype, 32, fpcr); end;
    end;

    return result[31:16];
end;
```

Library pseudocode for shared/functions/float/bfloat/BFSub

```
// BFSub()
// =====
// Non-widening BFloat16 subtraction used by SVE2 instructions.

func BFSub(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = TRUE;
    return BFSub(op1, op2, fpcr, fpexc);
end;

// BFSub()
// =====
// Non-widening BFloat16 subtraction following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 - op2.
// The 'fpcr' argument supplies the FPCR control bits.

func BFSub(op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type, fpexc : boolean) => bits(16)
begin
    let rounding : FPRounding = FPRoundingMode(fpcr);
    var done : boolean;
    var result : bits(32);

    let op1_s : bits(32) = op1 :: Zeros{16};
    let op2_s : bits(32) = op2 :: Zeros{16};
    let (type1, sign1, value1) = FPUnpack{32}(op1_s, fpcr, fpexc);
    let (type2, sign2, value2) = FPUnpack{32}(op2_s, fpcr, fpexc);

    (done, result) = FPProcessNaNs{32}(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        let inf1 = (type1 == FPType Infinity);
        let inf2 = (type2 == FPType Infinity);
        let zero1 = (type1 == FPType Zero);
        let zero2 = (type2 == FPType Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc InvalidOp, fpcr); end;
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity{32}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity{32}('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero{32}(sign1);
        else
            let result_value : real = value1 - value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding NEGINF then '1' else '0';
                result = FPZero{32}(result_sign);
            else
                result = FPRoundBF{32}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if fpexc then FPProcessDenorms(type1, type2, 32, fpcr); end;
    end;

    return result[31:16];
end;
```

Library pseudocode for shared/functions/float/bfloat/BFSub_ZA

```
// BFSub_ZA()
// =====
// Non-widening BFloat16 subtraction used by SME2 ZA-targeting instructions.

func BFSub_ZA(op1 : bits(16), op2 : bits(16), fpcr_in : FPCR_Type) => bits(16)
begin
    let fpexc : boolean = FALSE;
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFSub(op1, op2, fpcr, fpexc);
end;
```

Library pseudocode for shared/functions/float/bfloat/BFUnpack

```
// BFUnpack()
// =====
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

func BFUnpack{N}(fpval : bits(N)) => (FType, bit, real)
begin
  assert N IN {16,32};

  var sign : bit;
  var exp : bits(8);
  var frac : bits(23);
  if N == 16 then
    sign = fpval[15];
    exp = fpval[14:7];
    frac = fpval[6:0] :: Zeros{16};
  else // N == 32
    sign = fpval[31];
    exp = fpval[30:23];
    frac = fpval[22:0];
  end;

  var fptype : FType;
  var value : real;
  if IsZero(exp) then
    fptype = FType Zero; value = 0.0; // Fixed Flush to Zero
  elsif IsOnes(exp) then
    if IsZero(frac) then
      fptype = FType Infinity; value = 2.0^1000000;
    else // no SNaN for BF16 arithmetic
      fptype = FType QNaN; value = 0.0;
    end;
  else
    fptype = FType Nonzero;
    value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);
  end;

  if sign == '1' then value = -value; end;

  return (fptype, sign, value);
end;
```

Library pseudocode for shared/functions/float/bfloat/BFZero

```
// BFZero()
// =====

func BFZero{N}(sign : bit) => bits(N)
begin
  assert N == 16;
  let E : integer{} = 8;
  let F : integer{} = N - (E + 1);
  return sign :: Zeros{E} :: Zeros{F};
end;
```

Library pseudocode for shared/functions/float/bfloat/FPAdd_BF16

```
// FPAdd_BF16()
// =====
// Single-precision add following BFloat16 computation behaviors.

func FPAdd_BF16(op1 : bits(32), op2 : bits(32), fpcr : FPCR_Type) => bits(32)
begin
    var result : bits(32);

    let (type1,sign1,value1) = BFUnpack{32}(op1);
    let (type2,sign2,value2) = BFUnpack{32}(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN{32}(fpcr);
    else
        let inf1 = (type1 == FPType_Infinity);
        let inf2 = (type2 == FPType_Infinity);
        let zero1 = (type1 == FPType_Zero);
        let zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN{32}(fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity{32}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity{32}('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero{32}(sign1);
        else
            let result_value : real = value1 + value2;
            if result_value == 0.0 then
                result = FPZero{32}('0'); // Positive sign when Round to Odd
            else
                result = BFRound(result_value);
            end;
        end;
    end;
    return result;
end;
```

Library pseudocode for shared/functions/float/bfloat/FPConvertBF

```
// FPConvertBF()
// =====
// Converts a single-precision OP to BFloat16 value using the
// Round to Nearest Even rounding mode when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

func FPConvertBF(op : bits(32), fpcr_in : FPCR_Type, rounding_in : FPRounding) => bits(16)
begin
    let halfsize : integer{} = 16;
    var fpcr : FPCR_Type = fpcr_in;
    var rounding : FPRounding = rounding_in;
    var result : bits(32); // BF16 value in top 16 bits
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.[FIZ,FZ] = '11'; end; // Flush denormal input and output to zero
    if altfp then rounding = FPRounding\_TIEEVEN; end; // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    let (fptype,sign,value) = FPUnpack{32}(op, fpcr, fpexc);

    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN{32}(fpcr);
        else
            result = FPConvertNaN{32, 32}(op);
        end;
        if fptype == FPTYPE\_SNaN then
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr); end;
        end;
    elseif fptype == FPTYPE\_Infinity then
        result = FPInfinity{32}(sign);
    elseif fptype == FPTYPE\_Zero then
        result = FPZero{32}(sign);
    else
        result = FPRoundBF{32}(value, fpcr, rounding, fpexc);
    end;

    // Returns correctly rounded BF16 value from top 16 bits
    return result[(2*halfsize)-1:halfsize];
end;

// FPConvertBF()
// =====
// Converts a single-precision operand to BFloat16 value.

func FPConvertBF(op : bits(32), fpcr : FPCR_Type) => bits(16)
begin
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
end;
```

Library pseudocode for shared/functions/float/bfloat/FPRoundBF

```
// FPRoundBF()
// =====
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

func FPRoundBF{N}(op : real, fpcr : FPCR_Type, rounding : FPRounding, fpexc : boolean) => bits(N)
begin
    assert N == 32;
    let isbfloat16 : boolean = TRUE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc);
end;
```

Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// =====

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

func FixedToFP{N, M}(op : bits(M), fbits : integer, unsigned : boolean, fpcr : FPCR_Type,
    rounding : FPRounding) => bits(N)
begin
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    var result : bits(N);
    assert fbits >= 0;
    assert rounding != FPRounding\_ODD;

    // Correct signed-ness
    let int_operand : integer{} = if unsigned then UInt(op) else SInt(op);

    // Scale by fractional bits and generate a real value
    let real_operand : real = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero{N}('0');
    else
        result = FPRound{N}(real_operand, fpcr, rounding);
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fp8float/BFConvertFP8

```
// BFConvertFP8()
// =====
// Converts a BFloat16 OP to FP8 value.

func BFConvertFP8(op_in : bits(16), fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(8)
begin
    let op : bits(32) = op_in :: Zeros{16};
    return FPConvertFP8{8, 32}(op, fpcr, fpmr);
end;
```

Library pseudocode for shared/functions/float/fp8float/FP8Bits

```
// FP8Bits()
// =====
// Returns the minimum exponent, numbers of exponent and fraction bits.

func FP8Bits(fp8type : FP8Type) => FPBitsType
begin
    var minimum_exp : integer;
    var F : integer{2..3};
    if fp8type == FP8Type\_OFP8\_E4M3 then
        minimum_exp = -6; F = 3;
    else // fp8type == FP8Type_OFP8_E5M2
        minimum_exp = -14; F = 2;
    end;

    return (F, minimum_exp);
end;
```

Library pseudocode for shared/functions/float/fp8float/FP8ConvertBF

```
// FP8ConvertBF()
// =====
// Converts an FP8 operand to BFloat16 value.

func FP8ConvertBF(op : bits(8), issrc2 : boolean, fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(16)
begin
    let isbfloat16 : boolean = TRUE;
    let result : bits(32) = FP8ConvertFP{}(op, issrc2, fpcr, fpmr, isbfloat16);
    return result[16+:16];
end;
```

Library pseudocode for shared/functions/float/fp8float/FP8ConvertFP

```
// FP8ConvertFP()
// =====
// Converts an FP8 operand to half-precision value.

func FP8ConvertFP(op : bits(8), issrc2 : boolean, fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(16)
begin
    let isbfloat16 : boolean = FALSE;
    return FP8ConvertFP{16}(op, issrc2, fpcr, fpmr, isbfloat16);
end;

// FP8ConvertFP()
// =====
// Converts an FP8 operand to half-precision or BFloat16 value.
// The downscaling factor in FPMR.LSCALE or FPMR.LSCALE2 is applied to
// the value before rounding.

func FP8ConvertFP{M}(op : bits(8), issrc2 : boolean, fpcr_in : FPCR_Type, fpmr : FPMR_Type,
    isbfloat16 : boolean) => bits(M)
begin
    assert M IN {16,32};
    var result : bits(M);

    let fpexc : boolean = TRUE;
    var fpcr : FPCR_Type = fpcr_in;
    // Do not flush denormal inputs and outputs to zero.
    // Do not support alternative half-precision format.
    fpcr.[FI2,FZ,FZ16,AHP] = '0000';
    let rounding = FPRounding\_TIEEVEN;
    let fp8type : FP8Type = (if issrc2 then FP8DecodeType(fpmr.F8S2)
        else FP8DecodeType(fpmr.F8S1));

    let (fptype,sign,value) = FP8Unpack{8}(op, fp8type);

    if fptype == FPType\_SNaN || fptype == FPType\_QNaN then
        result = FPDefaultNaN{M}(fpcr);
        if fptype == FPType\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
        end;
    elsif fptype == FPType\_Infinity then
        result = FPInfinity{M}(sign);
    elsif fptype == FPType\_Zero then
        result = FPZero{M}(sign);
    else
        var dscale : integer;
        if issrc2 then
            dscale = (if M == 16 then UInt(fpmr.LSCALE2[3:0])
                else UInt(fpmr.LSCALE2[:6]));
        else
            dscale = (if M == 16 then UInt(fpmr.LSCALE[3:0])
                else UInt(fpmr.LSCALE[:6]));
        end;
        let result_value : real = value * (2.0^-dscale);
        result = FPRoundBase{M}(result_value, fpcr, rounding, isbfloat16, fpexc);
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fp8float/FP8DecodeType

```
// FP8DecodeType()
// =====
// Decode the FP8 format encoded in F8S1, F8S2 or F8D field in FPMR

func FP8DecodeType(f8format : bits(3)) => FP8Type
begin
    case f8format of
        when '000' => return FP8Type\_OFP8\_E5M2;
        when '001' => return FP8Type\_OFP8\_E4M3;
        otherwise => return FP8Type\_UNSUPPORTED;
    end;
end;
```

Library pseudocode for shared/functions/float/fp8float/FP8DefaultNaN

```
// FP8DefaultNaN()
// =====

func FP8DefaultNaN{N}(fp8type : FP8Type, fpcr : FPCR_Type) => bits(N)
begin
  assert N == 8;
  assert fp8type IN {FP8Type OFP8 E5M2, FP8Type OFP8 E4M3};
  let sign : bit = if IsFeatureImplemented(FEAT_AFP) then fpcr.AH else '0';
  let E : integer{} = if fp8type == FP8Type OFP8 E4M3 then 4 else 5;
  let F : integer{} = N - (E + 1);
  var exp : bits(E);
  var frac : bits(F);

  case fp8type of
    when FP8Type OFP8 E4M3 =>
      exp = Ones{E};
      frac = Ones{F};
    when FP8Type OFP8 E5M2 =>
      exp = Ones{E};
      frac = '1'::Zeros{F-1};
  end;

  return sign :: exp :: frac;
end;
```



```

// FP8DotAddFP()
// =====

func FP8DotAddFP(M, N)(addend : bits(M), op1 : bits(N), op2 : bits(N),
    fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(M)

begin
    let E : integer{} = (N DIV 8) as integer{1, 2, 4, 8};
    return FP8DotAddFP{M, N}(addend, op1, op2, E, fpcr, fpmr);
end;

// FP8DotAddFP()
// =====
// Calculates result of "E"-way 8-bit floating-point dot-product with scaling
// and addition to half-precision or single-precision value without
// intermediate rounding.
// c = round(c + 2-S*(a1*b1+...+aE*bE))
// The 8-bit floating-point format for op1 is determined by FPMR.F8S1
// and the one for op2 by FPMR.F8S2. The scaling factor in FPMR.LSCALE
// is applied to the sum-of-products before adding to the addend and rounding.

func FP8DotAddFP(M, N)(addend : bits(M), op1 : bits(N), op2 : bits(N), E : integer{1, 2, 4, 8},
    fpcr_in : FPCR_Type, fpmr : FPMR_Type) => bits(M)

begin
    assert M IN {16,32};
    assert N IN {2*M, M, M DIV 2, M DIV 4};
    var fpcr : FPCR_Type = fpcr_in;
    var result : bits(M);

    fpcr.[FIZ,FZ,FZ16] = '000'; // Do not flush denormal inputs and outputs to zero
    fpcr.DN = '1';
    let rounding = FPRounding\_TIEEVEN;

    let fp8type1 : FP8Type = FP8DecodeType(fpmr.F8S1);
    let fp8type2 : FP8Type = FP8DecodeType(fpmr.F8S2);

    var type1 : array[[E]] of FPType;
    var type2 : array[[E]] of FPType;
    var sign1 : array[[E]] of bit;
    var sign2 : array[[E]] of bit;
    var value1 : array[[E]] of real;
    var value2 : array[[E]] of real;
    var inf1 : array[[E]] of boolean;
    var inf2 : array[[E]] of boolean;
    var zero1 : array[[E]] of boolean;
    var zero2 : array[[E]] of boolean;

    let fpexc : boolean = FALSE;
    let (typeA,signA,valueA) = FPUnpack{M}(addend, fpcr, fpexc);
    let infA = (typeA == FPType\_Infinity); let zeroA = (typeA == FPType\_Zero);
    var any_nan : boolean = typeA IN {FPType\_SNaN, FPType\_QNaN};
    for i = 0 to E-1 do
        (type1[[i]], sign1[[i]], value1[[i]]) = FP8Unpack{N DIV E}(op1[i*:(N DIV E)], fp8type1);
        (type2[[i]], sign2[[i]], value2[[i]]) = FP8Unpack{N DIV E}(op2[i*:(N DIV E)], fp8type2);
        inf1[[i]] = (type1[[i]] == FPType\_Infinity); zero1[[i]] = (type1[[i]] == FPType\_Zero);
        inf2[[i]] = (type2[[i]] == FPType\_Infinity); zero2[[i]] = (type2[[i]] == FPType\_Zero);
        any_nan = (any_nan || type1[[i]] IN {FPType\_SNaN, FPType\_QNaN} ||
            type2[[i]] IN {FPType\_SNaN, FPType\_QNaN});
    end;

    if any_nan then
        result = FPDefaultNaN{M}(fpcr);
    else
        // Determine sign and type products will have if it does not cause an Invalid
        // Operation.
        var signP : array [[E]] of bit;
        var infP : array [[E]] of boolean;
        var zeroP : array [[E]] of boolean;
        for i = 0 to E-1 do
            signP[[i]] = sign1[[i]] XOR sign2[[i]];
            infP[[i]] = inf1[[i]] || inf2[[i]];
            zeroP[[i]] = zero1[[i]] || zero2[[i]];
        end;

        // Detect non-numeric results of dot product and accumulate
        var posInfR : boolean = (infA && signA == '0');
        var negInfR : boolean = (infA && signA == '1');
        var zeroR : boolean = zeroA;
        var invalidop : boolean = FALSE;
        for i = 0 to E-1 do
            // Result is infinity if any input is infinity
            posInfR = posInfR || (infP[[i]] && signP[[i]] == '0');
            negInfR = negInfR || (infP[[i]] && signP[[i]] == '1');
            // Result is zero if the addend and the products are zeroes of the same sign
            zeroR = zeroR && zeroP[[i]] && (signA == signP[[i]]);
            // Non SNaN-generated Invalid Operation cases are multiplies of zero
            // by infinity and additions of opposite-signed infinities.
            invalidop = (invalidop || (inf1[[i]] && zero2[[i]]) || (zero1[[i]] && inf2[[i]]) ||
                (infA && infP[[i]] && (signA != signP[[i]])));
            for j = i+1 to E-1 do

```

```

        invalidop = invalidop || (infP[[i]] && infP[[j]] && (signP[[i]] != signP[[j]]));
    end;
end;

if invalidop then
    result = FPDefaultNaN{M}(fpcr);

    // Other cases involving infinities produce an infinity of the same sign.
elseif posInfR then
    result = FPInfinity{M}('0');
elseif negInfR then
    result = FPInfinity{M}('1');

    // Cases where the result is exactly zero and its sign is not determined by the
    // rounding mode are additions of same-signed zeros.
elseif zeroR then
    result = FPZero{M}(signA);

    // Otherwise calculate numerical value and round it.
else
    // Apply scaling to sum-of-product
    let dscale : integer = if M == 32 then UInt(fpmr.LSCALE) else UInt(fpmr.LSCALE[3:0]);

    var dp_value : real = value1[[0]] * value2[[0]];
    for i = 1 to E-1 do
        dp_value = dp_value + value1[[i]] * value2[[i]];
    end;

    let result_value : real = valueA + dp_value * (2.0^(-dscale));
    if result_value == 0.0 then // Sign of exact zero result is '0' for RNE rounding mode
        result = FPZero{M}('0');
    else
        let satoflo : boolean = (fpmr.OSM == '1');
        result = FPRound FP8{M}(result_value, fpcr, rounding, satoflo);
    end;
end;
end;

return result;
end;

```

Library pseudocode for shared/functions/float/fp8float/FP8Infinity

```

// FP8Infinity()
// =====

func FP8Infinity(N)(fp8type : FP8Type, sign : bit) => bits(N)
begin
    assert N == 8;
    assert fp8type IN {FP8Type OFP8 E5M2, FP8Type OFP8 E4M3};
    let E : integer{} = if fp8type == FP8Type OFP8 E4M3 then 4 else 5;
    let F : integer{} = N - (E + 1);
    var exp : bits(E);
    var frac : bits(F);

    case fp8type of
        when FP8Type OFP8 E4M3 =>
            exp = Ones{E};
            frac = Ones{F};
        when FP8Type OFP8 E5M2 =>
            exp = Ones{E};
            frac = Zeros{F};
    end;

    return sign :: exp :: frac;
end;

```

Library pseudocode for shared/functions/float/fp8float/FP8MatMulAddFP

```
// FP8MatMulAddFP()
// =====
// 8-bit floating-point matrix multiply with scaling and add to half-precision
// or single-precision matrix.
// result[2, 2] = addend[2, 2] + (op1[2, E] * op2[E, 2])

func FP8MatMulAddFP{N}(addend : bits(N), op1 : bits(N), op2 : bits(N), E : integer{4, 8},
    fpcr : FPCR_Type, fpmr : FPMR_Type) => bits(N)
begin
    assert N IN {64, 128};
    assert N == E*16;
    let M : integer{} = N DIV 4;
    var result : bits(N);

    for i = 0 to 1 do
        for j = 0 to 1 do
            let elt1 : bits(2*M) = op1[i*:(2*M)];
            let elt2 : bits(2*M) = op2[j*:(2*M)];
            let sum : bits(M) = addend[(2*i + j)*:M];
            result[(2*i + j)*:M] = FP8DotAddFP{M, N DIV 2}(sum, elt1, elt2, E, fpcr, fpmr);
        end;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fp8float/FP8MaxNormal

```
// FP8MaxNormal()
// =====

func FP8MaxNormal{N}(fp8type : FP8Type, sign : bit) => bits(N)
begin
    assert N == 8;
    assert fp8type IN {FP8Type OFP8 E5M2, FP8Type OFP8 E4M3};
    let E : integer{} = if fp8type == FP8Type OFP8 E4M3 then 4 else 5;
    let F : integer{} = N - (E + 1);
    var exp : bits(E);
    var frac : bits(F);

    case fp8type of
        when FP8Type OFP8 E4M3 =>
            exp = Ones{E};
            frac = Ones{F-1}::'0';
        when FP8Type OFP8 E5M2 =>
            exp = Ones{E-1}::'0';
            frac = Ones{F};
    end;

    return sign :: exp :: frac;
end;
```

Library pseudocode for shared/functions/float/fp8float/FP8MulAddFP

```
// FP8MulAddFP()
// =====

func FP8MulAddFP{M}(addend : bits(M), op1 : bits(8), op2 : bits(8), fpcr : FPCR_Type,
    fpmr : FPMR_Type) => bits(M)
begin
    assert M IN {16, 32};
    let E : integer{} = 1;
    return FP8DotAddFP{M, 8}(addend, op1, op2, E, fpcr, fpmr);
end;
```



```

// FP8Round()
// =====
// Used by FP8 downconvert instructions which observe FPMR.OSC
// to convert a real number OP into an FP8 value.

func FP8Round(N)(op : real, fp8type : FP8Type, fpcr : FPCR\_Type, fpmr : FPMR\_Type) => bits(N)
begin
    assert N == 8;
    assert fp8type IN {FP8Type OFP8 E5M2, FP8Type OFP8 E4M3};
    assert op != 0.0;
    var result : bits(N);

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    let (F, minimum_exp) = FP8Bits(fp8type);
    let E : integer{} = (N - F) - 1;

    // Split value into sign, unrounded mantissa and exponent.
    var sign : bit;
    var exponent : integer;
    var mantissa : real;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    end;

    (mantissa, exponent) = NormalizeReal(mantissa);
    // When TRUE, detection of underflow occurs after rounding.
    let altfp : boolean = IsFeatureImplemented(FEAT_AFF) && fpcr.AH == '1';

    var biased_exp_unconstrained : integer = (exponent - minimum_exp) + 1;
    var int_mant_unconstrained : integer = RoundDown(mantissa * 2.0^F);
    let error_unconstrained : real = mantissa * 2.0^F - Real(int_mant_unconstrained);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    var biased_exp : integer = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent); end;

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    var int_mant : integer = RoundDown(mantissa * 2.0^F);
    var error : real = mantissa * 2.0^F - Real(int_mant);

    let trapped_UF : boolean = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());

    var round_up_unconstrained : boolean;
    var round_up : boolean;

    if altfp then
        // Round to Nearest Even
        round_up_unconstrained = (error_unconstrained > 0.5 ||
            (error_unconstrained == 0.5 && int_mant_unconstrained[0] == '1'));
        round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));

        if round_up_unconstrained then
            int_mant_unconstrained = int_mant_unconstrained + 1;
            if int_mant_unconstrained == 2^(F+1) then // Rounded up to next exponent
                biased_exp_unconstrained = biased_exp_unconstrained + 1;
                int_mant_unconstrained = int_mant_unconstrained DIV 2;
            end;
        end;

        // Follow alternate floating-point behavior of underflow after rounding
        if (biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 &&
            (error != 0.0 || trapped_UF)) then
            FPProcessException(FPExc Underflow, fpcr);
        end;
    else // altfp == FALSE
        // Underflow occurs if exponent is too small before rounding, and result is inexact or
        // the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
        if biased_exp == 0 && (error != 0.0 || trapped_UF) then
            FPProcessException(FPExc Underflow, fpcr);
        end;

        // Round to Nearest Even
        round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));
    end;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then // Rounded up from denormalized to normalized
            biased_exp = 1;
        end;
        if int_mant == 2^(F+1) then // Rounded up to next exponent
            biased_exp = biased_exp + 1;
            int_mant = int_mant DIV 2;
        end;
    end;
end;

```

```

// Deal with overflow and generate result.
var overflow : boolean;
case fp8type of
  when FP8Type\_OFP8\_E4M3 =>
    overflow = biased_exp >= 2^E || (biased_exp == 2^E - 1 && int_mant == 2^(F+1) - 1);
  when FP8Type\_OFP8\_E5M2 =>
    overflow = biased_exp >= 2^E - 1;
end;

if overflow then
  result = (if fpmr.OSC == '0' then FP8Infinity{N}(fp8type, sign)
    else FP8MaxNormal{N}(fp8type, sign));
  // Flag Overflow exception regardless of FPMR.OSC
  FPPProcessException(FPExc\_Overflow, fpcr);
  error = 1.0; // Ensure that an Inexact exception occurs
else
  result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0];
end;

// Deal with Inexact exception.
if error != 0.0 then
  FPPProcessException(FPExc\_Inexact, fpcr);
end;

return result;
end;

```

Library pseudocode for shared/functions/float/fp8float/FP8Type

```

// FP8Type
// =====

type FP8Type of enumeration {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3, FP8Type_UNSUPPORTED};

```

Library pseudocode for shared/functions/float/fp8float/FP8Unpack

```
// FP8Unpack()
// =====
// Unpacks an FP8 value into its type, sign bit and real number that
// it represents.

func FP8Unpack{N}(fpval : bits(N), fp8type : FP8Type) => (FType, bit, real)
begin
  assert N == 8;
  let E : integer{} = if fp8type == FP8Type\_OFP8\_E4M3 then 4 else 5;
  let F : integer{} = N - (E + 1);

  let sign : bit = fpval[N-1];
  let exp : bits(E) = fpval[(E+F)-1:F];
  let frac : bits(F) = fpval[F-1:0];

  var value : real;
  var fptype : FType;

  if fp8type == FP8Type\_OFP8\_E4M3 then
    if IsZero(exp) then
      if IsZero(frac) then
        fptype = FType\_Zero; value = 0.0;
      else
        fptype = FType\_Denormal; value = 2.0^-6 * (Real(UInt(frac)) * 2.0^-3);
      end;
    elsif IsOnes(exp) && IsOnes(frac) then
      fptype = FType\_SNaN;
      value = 0.0;
    else
      fptype = FType\_Nonzero;
      value = 2.0^(UInt(exp)-7) * (1.0 + Real(UInt(frac)) * 2.0^-3);
    end;

  elsif fp8type == FP8Type\_OFP8\_E5M2 then
    if IsZero(exp) then
      if IsZero(frac) then
        fptype = FType\_Zero; value = 0.0;
      else
        fptype = FType\_Denormal; value = 2.0^-14 * (Real(UInt(frac)) * 2.0^-2);
      end;
    elsif IsOnes(exp) then
      if IsZero(frac) then
        fptype = FType\_Infinity; value = 2.0^1000000;
      else
        fptype = if frac[1] == '1' then FType\_QNaN else FType\_SNaN;
        value = 0.0;
      end;
    else
      fptype = FType\_Nonzero;
      value = 2.0^(UInt(exp)-15) * (1.0 + Real(UInt(frac)) * 2.0^-2);
    end;

  else // fp8type == FP8Type_UNSUPPORTED
    fptype = FType\_SNaN;
    value = 0.0;
  end;

  if sign == '1' then value = -value; end;

  return (fptype, sign, value);
end;
```

Library pseudocode for shared/functions/float/fp8float/FP8Zero

```
// FP8Zero()
// =====

func FP8Zero{N}(fp8type : FP8Type, sign : bit) => bits(N)
begin
  assert N == 8;
  assert fp8type IN {FP8Type\_OFP8\_E5M2, FP8Type\_OFP8\_E4M3};
  let E : integer{} = if fp8type == FP8Type\_OFP8\_E4M3 then 4 else 5;
  let F : integer{} = N - (E + 1);
  return sign :: Zeros{E} :: Zeros{F};
end;
```


Library pseudocode for shared/functions/float/fp8float/FPConvertFP8

```
// FPConvertFP8()
// =====
// Converts a half-precision or single-precision OP to FP8 value.
// The scaling factor in FPMR.NSCALE is applied to the value before rounding.

func FPConvertFP8{M, N}(op : bits(N), fpcr_in : FPCR_Type, fpmr : FPMR_Type) => bits(M)
begin
  assert N IN {16,32} && M == 8;
  var result : bits(M);

  let fpexc : boolean = TRUE;
  var fpcr : FPCR_Type = fpcr_in;
  fpcr.[FIZ,FZ,FZ16] = '000'; // Do not flush denormal inputs and outputs to zero
  let fp8type : FP8Type = FP8DecodeType(fpmr.F8D);

  let (fptype,sign,value) = FPUnpack{N}(op, fpcr, fpexc);

  if fp8type == FP8Type\_UNSUPPORTED then
    result = Ones{M};
    FPProcessException(FPExc\_InvalidOp, fpcr);
  elsif fptype == FType\_SNaN || fptype == FType\_QNaN then
    result = FP8DefaultNaN{M}(fp8type, fpcr); // Always generate Default NaN as result
    if fptype == FType\_SNaN then
      FPProcessException(FPExc\_InvalidOp, fpcr);
    end;
  elsif fptype == FType\_Infinity then
    result = (if fpmr.OSC == '0' then FP8Infinity{M}(fp8type, sign)
              else FP8MaxNormal{M}(fp8type, sign));
  elsif fptype == FType\_Zero then
    result = FP8Zero{M}(fp8type, sign);
  else
    let scale : integer = if N == 16 then SInt(fpmr.NSCALE[4:0]) else SInt(fpmr.NSCALE);
    let result_value : real = value * (2.0^scale);
    result = FP8Round{M}(result_value, fp8type, fpcr, fpmr);
  end;

  return result;
end;
```

Library pseudocode for shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =====

func FPAbs{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  assert N IN {16,32,64};
  if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
    if fpcr.AH == '1' then
      let (fptype, -, -) = FPUnpack{N}(op, fpcr, FALSE);
      if fptype IN {FType\_SNaN, FType\_QNaN} then
        return op; // When fpcr.AH=1, sign of NaN has no consequence
      end;
    end;
  end;
  return '0' :: op[N-2:0];
end;
```

Library pseudocode for shared/functions/float/fpabsmax/FPAbsMax

```
// FPAbsMax()
// =====
// Compare absolute value of two operands and return the larger absolute
// value without rounding.

func FPAbsMax{N}(op1_in : bits(N), op2_in : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.[AH,FIZ,FZ,FZ16] = '0000';

    let op1 = '0'::op1_in[N-2:0];
    let op2 = '0'::op2_in[N-2:0];
    let (type1,-,value1) = FPUnpack{N}(op1, fpcr);
    let (type2,-,value2) = FPUnpack{N}(op2, fpcr);

    (done,result) = FPProcessNaNs{N}(type1, type2, op1_in, op2_in, fpcr);

    if !done then
        // This condition covers all results other than NaNs,
        // including Zero & Infinity
        result = if value1 > value2 then op1 else op2;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpabsmin/FPAbsMin

```
// FPAbsMin()
// =====
// Compare absolute value of two operands and return the smaller absolute
// value without rounding.

func FPAbsMin{N}(op1_in : bits(N), op2_in : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.[AH,FIZ,FZ,FZ16] = '0000';

    let op1 = '0'::op1_in[N-2:0];
    let op2 = '0'::op2_in[N-2:0];
    let (type1,-,value1) = FPUnpack{N}(op1, fpcr);
    let (type2,-,value2) = FPUnpack{N}(op2, fpcr);

    (done,result) = FPProcessNaNs{N}(type1, type2, op1_in, op2_in, fpcr);

    if !done then
        // This condition covers all results other than NaNs,
        // including Zero & Infinity
        result = if value1 < value2 then op1 else op2;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =====

func FPAdd{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;          // Generate floating-point exceptions
    return FPAdd{N}(op1, op2, fpcr, fpexc);
end;

// FPAdd()
// =====

func FPAdd{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    let rounding : FPRounding = FPRoundingMode(fpcr);

    let (type1,sign1,value1) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) = FPUnpack{N}(op2, fpcr, fpexc);

    var (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        let inf1 : boolean = (type1 == FPType\_Infinity);
        let inf2 : boolean = (type2 == FPType\_Infinity);
        let zero1 : boolean = (type1 == FPType\_Zero);
        let zero2 : boolean = (type2 == FPType\_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN{N}(fpcr);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr); end;
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity{N}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity{N}('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero{N}(sign1);
        else
            let result_value : real = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero{N}(result_sign);
            else
                result = FPRound{N}(result_value, fpcr, rounding, fpexc);
            end;
        end;
    end;

    if fpexc then FPProcessDenorms(type1, type2, N, fpcr); end;
end;
return result;
end;
```

Library pseudocode for shared/functions/float/fpadd/FPAdd_ZA

```
// FPAdd_ZA()
// =====
// Calculates op1+op2 for SME2 ZA-targeting instructions.

func FPAdd_ZA{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    let fpexc : boolean = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';                // Generate default NaN values
    return FPAdd{N}(op1, op2, fpcr, fpexc);
end;
```

Library pseudocode for shared/functions/float/fpbits/FPBits

```
// FPBits()
// =====
// Returns the minimum exponent, numbers of exponent and fraction bits.

func FPBits(N : integer, isbfloat16 : boolean) => FPBitsType
begin
  var F : FPFracBits;
  var minimum_exp : integer;
  if N == 16 then
    minimum_exp = -14; F = 10;
  elsif N == 32 && isbfloat16 then
    minimum_exp = -126; F = 7;
  elsif N == 32 then
    minimum_exp = -126; F = 23;
  else // N == 64
    minimum_exp = -1022; F = 52;
  end;

  return (F, minimum_exp);
end;
```

Library pseudocode for shared/functions/float/fpbits/FPBitsType

```
// FPBitsType
// =====

type FPBitsType of (FPFracBits, integer);
```

Library pseudocode for shared/functions/float/fpbits/FPFracBits

```
// FPFracBits
// =====

type FPFracBits of integer{2, 3, 7, 10, 23, 52};
```

Library pseudocode for shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// =====

func FPCompare{N}(op1 : bits(N), op2 : bits(N), signal_nans : boolean, fpcr : FPCR\_Type) => bits(4)
begin
  assert N IN {16,32,64};
  let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
  let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);

  var result : bits(4);
  if type1 IN {FPType\_SNaN, FPType\_QNaN} || type2 IN {FPType\_SNaN, FPType\_QNaN} then
    result = '0011';
    if type1 == FPType\_SNaN || type2 == FPType\_SNaN || signal_nans then
      FPProcessException(FPExc\_InvalidOp, fpcr);
    end;
  else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    if value1 == value2 then
      result = '0110';
    elsif value1 < value2 then
      result = '1000';
    else // value1 > value2
      result = '0010';
    end;

    FPProcessDenorms(type1, type2, N, fpcr);
  end;
  return result;
end;
```

Library pseudocode for shared/functions/float/fpcompareeq/FPCmpareEQ

```
// FPCmpareEQ()
// =====

func FPCmpareEQ{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
  assert N IN {16,32,64};
  let (type1,sign1,value1) : (FType, bit, real) = FPUntpack{N}(op1, fpcr);
  let (type2,sign2,value2) : (FType, bit, real) = FPUntpack{N}(op2, fpcr);

  var result : boolean;
  if type1 IN {FType_SNaN, FType_QNaN} || type2 IN {FType_SNaN, FType_QNaN} then
    result = FALSE;
    if type1 == FType_SNaN || type2 == FType_SNaN then
      FFPProcessException(FPExc_InvalidOp, fpcr);
    end;
  else
    // All non-NaN cases can be evaluated on the values produced by FPUntpack()
    result = (value1 == value2);
    FFPProcessDenorms(type1, type2, N, fpcr);

  end;
  return result;
end;
```

Library pseudocode for shared/functions/float/fpcomparege/FPCmpareGE

```
// FPCmpareGE()
// =====

func FPCmpareGE{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
  assert N IN {16,32,64};
  let (type1,sign1,value1) : (FType, bit, real) = FPUntpack{N}(op1, fpcr);
  let (type2,sign2,value2) : (FType, bit, real) = FPUntpack{N}(op2, fpcr);

  var result : boolean;
  if type1 IN {FType_SNaN, FType_QNaN} || type2 IN {FType_SNaN, FType_QNaN} then
    result = FALSE;
    FFPProcessException(FPExc_InvalidOp, fpcr);
  else
    // All non-NaN cases can be evaluated on the values produced by FPUntpack()
    result = (value1 >= value2);
    FFPProcessDenorms(type1, type2, N, fpcr);

  end;
  return result;
end;
```

Library pseudocode for shared/functions/float/fpcomparegt/FPCmpareGT

```
// FPCmpareGT()
// =====

func FPCmpareGT{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
  assert N IN {16,32,64};
  let (type1,sign1,value1) : (FType, bit, real) = FPUntpack{N}(op1, fpcr);
  let (type2,sign2,value2) : (FType, bit, real) = FPUntpack{N}(op2, fpcr);

  var result : boolean;
  if type1 IN {FType_SNaN, FType_QNaN} || type2 IN {FType_SNaN, FType_QNaN} then
    result = FALSE;
    FFPProcessException(FPExc_InvalidOp, fpcr);
  else
    // All non-NaN cases can be evaluated on the values produced by FPUntpack()
    result = (value1 > value2);
    FFPProcessDenorms(type1, type2, N, fpcr);

  end;
  return result;
end;
```

Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// =====

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

func FPConvert{M, N}(op : bits(N), fpcr : FPCR_Type, rounding : FPRounding) => bits(M)
begin

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    var result : bits(M);

    // Unpack floating-point operand optionally with flush-to-zero.
    let (fptype,sign,value) : (FPTYPE, bit, real) = FPUnpackCV{N}(op, fpcr);

    let alt_hp : boolean = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        if alt_hp then
            result = FPZero{M}(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN{M}(fpcr);
        else
            result = FPConvertNaN{M, N}(op);
        end;
        if fptype == FPTYPE\_SNaN || alt_hp then
            FPProcessException(FPExc\_InvalidOp, fpcr);
        end;
    elsif fptype == FPTYPE\_Infinity then
        if alt_hp then
            result = sign::Ones{M-1};
            FPProcessException(FPExc\_InvalidOp, fpcr);
        else
            result = FPInfinity{M}(sign);
        end;
    elsif fptype == FPTYPE\_Zero then
        result = FPZero{M}(sign);
    else
        result = FPRoundCV{M}(value, fpcr, rounding);
        FPProcessDenorm(fptype, N, fpcr);
    end;
    return result;
end;

// FPConvert()
// =====

func FPConvert{M, N}(op : bits(N), fpcr : FPCR_Type) => bits(M)
begin
    return FPConvert{M, N}(op, fpcr, FPRoundingMode(fpcr));
end;
```

Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// =====
// Converts a NaN of one floating-point type to another

func FPConvertNaN{M, N}(op : bits(N)) => bits(M)
begin
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    var result : bits(M);
    var frac : bits(51);

    let sign : bit = op[N-1];

    // Unpack payload from input NaN
    case N of
        when 64 => frac = op[50:0];
        when 32 => frac = op[21:0]::Zeros{29};
        when 16 => frac = op[8:0]::Zeros{42};
    end;

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 => result = sign::Ones{M-52}::frac;
        when 32 => result = sign::Ones{M-23}::frac[50:29];
        when 16 => result = sign::Ones{M-10}::frac[50:42];
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// =====
// Decode most common AArch32 floating-point rounding encoding.

func FPDecodeRM(rm : bits(2)) => FPRounding
begin
    var result : FPRounding;
    case rm of
        when '00' => result = FPRounding\_TIEAWAY; // A
        when '01' => result = FPRounding\_TIEEVEN; // N
        when '10' => result = FPRounding\_POSINF; // P
        when '11' => result = FPRounding\_NEGINF; // M
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// =====
// Decode floating-point rounding mode and common AArch64 encoding.

func FPDecodeRounding(rmode : bits(2)) => FPRounding
begin
    case rmode of
        when '00' => return FPRounding\_TIEEVEN; // N
        when '01' => return FPRounding\_POSINF; // P
        when '10' => return FPRounding\_NEGINF; // M
        when '11' => return FPRounding\_ZERO; // Z
    end;
end;
```

Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// =====

func FPDefaultNaN{N}(fpcr : FPCR_Type) => bits(N)
begin
  assert N IN {16,32,64};
  let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
  let F : integer{} = N - (E + 1);
  let sign : bit = if IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() then fpcr.AH else '0';

  let exp : bits(E) = Ones{};
  let frac : bits(F) = '1'::Zeros{F-1};
  return sign :: exp :: frac;
end;
```

Library pseudocode for shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =====

func FPDiv{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  assert N IN {16,32,64};
  let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr);
  let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr);
  var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);

  if !done then
    let inf1 : boolean = type1 == FType Infinity;
    let inf2 : boolean = type2 == FType Infinity;
    let zero1 : boolean = type1 == FType Zero;
    let zero2 : boolean = type2 == FType Zero;

    if (inf1 && inf2) || (zero1 && zero2) then
      result = FPDefaultNaN{N}(fpcr);
      FPProcessException(FPExc InvalidOp, fpcr);
    elsif inf1 || zero2 then
      result = FPinfinity{N}(sign1 XOR sign2);
      if !inf1 then FPProcessException(FPExc DivideByZero, fpcr); end;
    elsif zero1 || inf2 then
      result = FPZero{N}(sign1 XOR sign2);
    else
      result = FPRound{N}(value1/value2, fpcr);
    end;

    if !zero2 then
      FPProcessDenorms(type1, type2, N, fpcr);
    end;
  end;
  return result;
end;
```


Library pseudocode for shared/functions/float/fpdot/FPDot

```
// FPDot()
// =====
// Calculates single-precision result of 2-way 16-bit floating-point dot-product
// with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'isbfloat16'
// determines whether input operands are BFloat16 or half-precision type.
// and 'fpexc' controls the generation of floating-point exceptions.

func FPDot(op1_a : bits(16), op1_b : bits(16), op2_a : bits(16),
           op2_b : bits(16), fpcr : FPCR_Type, isbfloat16 : boolean) => bits(32)
begin
    let fpexc : boolean = TRUE;           // Generate floating-point exceptions
    return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
end;

func FPDot(op1_a : bits(16), op1_b : bits(16), op2_a : bits(16),
           op2_b : bits(16), fpcr_in : FPCR_Type, isbfloat16 : boolean, fpexc : boolean) => bits(32)
begin
    var fpcr : FPCR_Type = fpcr_in;
    var result : bits(32);
    var done : boolean;
    fpcr.AHP = '0';           // Ignore alternative half-precision option
    let rounding : FPRounding = FPRoundingMode(fpcr);

    let (type1_a, sign1_a, value1_a) = FPUnpackBase{16}(op1_a, fpcr, fpexc, isbfloat16);
    let (type1_b, sign1_b, value1_b) = FPUnpackBase{16}(op1_b, fpcr, fpexc, isbfloat16);
    let (type2_a, sign2_a, value2_a) = FPUnpackBase{16}(op2_a, fpcr, fpexc, isbfloat16);
    let (type2_b, sign2_b, value2_b) = FPUnpackBase{16}(op2_b, fpcr, fpexc, isbfloat16);

    let inf1_a = (type1_a == FPTYPE\_Infinity); let zero1_a = (type1_a == FPTYPE\_Zero);
    let inf1_b = (type1_b == FPTYPE\_Infinity); let zero1_b = (type1_b == FPTYPE\_Zero);
    let inf2_a = (type2_a == FPTYPE\_Infinity); let zero2_a = (type2_a == FPTYPE\_Zero);
    let inf2_b = (type2_b == FPTYPE\_Infinity); let zero2_b = (type2_b == FPTYPE\_Zero);

    (done, result) = FPProcessNaNs4(type1_a, type1_b, type2_a, type2_b,
                                   op1_a, op1_b, op2_a, op2_b, fpcr, fpexc);

    if !done then
        // Determine sign and type products will have if it does not cause an Invalid
        // Operation.
        let signPa = sign1_a XOR sign2_a;
        let signPb = sign1_b XOR sign2_b;
        let infPa = inf1_a || inf2_a;
        let infPb = inf1_b || inf2_b;
        let zeroPa = zero1_a || zero2_a;
        let zeroPb = zero1_b || zero2_b;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        let invalidop = ((inf1_a && zero2_a) || (zero1_a && inf2_a) ||
                        (inf1_b && zero2_b) || (zero1_b && inf2_b) ||
                        (infPa && infPb && signPa != signPb));

        if invalidop then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr); end;

        // Other cases involving infinities produce an infinity of the same sign.
        elseif (infPa && signPa == '0') || (infPb && signPb == '0') then
            result = FPInfinity{32}('0');
        elseif (infPa && signPa == '1') || (infPb && signPb == '1') then
            result = FPInfinity{32}('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elseif zeroPa && zeroPb && signPa == signPb then
            result = FPZero{32}(signPa);

        // Otherwise calculate fused sum of products and round it.
        else
            let result_value = (value1_a * value2_a) + (value1_b * value2_b);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                let result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero{32}(result_sign);
            else
                result = FPRound{32}(result_value, fpcr, rounding, fpexc);
            end;
        end;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpdot/FPDotAdd

```
// FPDotAdd()
// =====
// Half-precision 2-way dot-product and add to single-precision.

func FPDotAdd(addend : bits(32), op1_a : bits(16), op1_b : bits(16),
              op2_a : bits(16), op2_b : bits(16), fpcr : FPCR_Type) => bits(32)
begin
  var prod : bits(32);
  let isbfloat16 : boolean = FALSE;
  let fpexc : boolean      = TRUE;      // Generate floating-point exceptions
  prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
  let result : bits(32) = FPAdd{}(addend, prod, fpcr, fpexc);

  return result;
end;
```

Library pseudocode for shared/functions/float/fpdot/FPDotAdd_ZA

```
// FPDotAdd_ZA()
// =====
// Half-precision 2-way dot-product and add to single-precision
// for SME ZA-targeting instructions.

func FPDotAdd_ZA(addend : bits(32), op1_a : bits(16), op1_b : bits(16),
                 op2_a : bits(16), op2_b : bits(16), fpcr_in : FPCR_Type) => bits(32)
begin
  var fpcr : FPCR_Type = fpcr_in;
  var prod : bits(32);
  let isbfloat16 : boolean = FALSE;
  let fpexc : boolean      = FALSE; // Do not generate floating-point exceptions
  fpcr.DN = '1';                // Generate default NaN values
  prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
  let result : bits(32) = FPAdd{}(addend, prod, fpcr, fpexc);

  return result;
end;
```

Library pseudocode for shared/functions/float/fpexc/FPExc

```
// FPExc
// =====

type FPExc of enumeration {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                          FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```
// FPIfinity()
// =====

func FPIfinity{N}(sign : bit) => bits(N)
begin
  assert N IN {16,32,64};
  let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
  let F : integer{} = N - (E + 1);
  let exp : bits(E) = Ones{};
  let frac : bits(F) = Zeros{};
  return sign :: exp :: frac;
end;
```

Library pseudocode for shared/functions/float/fpmatmul/FPMatMulAdd

```
// FPMatMulAdd()
// =====
//
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

func FPMatMulAdd{N}(addend : bits(N), op1 : bits(N), op2 : bits(N),
    esize : ESize, fpcr : FPCR_Type) => bits(N)
begin
    assert N == esize * 2 * 2;
    var result : bits(N);
    var prod0, prod1, sum : bits(esize);

    for i = 0 to 1 do
        for j = 0 to 1 do
            sum = addend[(2*i + j)*esize];
            prod0 = FPMul(esize){op1[(2*i + 0)*esize],
                op2[(2*j + 0)*esize], fpcr);
            prod1 = FPMul(esize){op1[(2*i + 1)*esize],
                op2[(2*j + 1)*esize], fpcr);
            sum = FPAdd(esize){sum, FPAdd(esize){prod0, prod1, fpcr}, fpcr);
            result[(2*i + j)*esize] = sum;
        end;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpmatmulh/FPMatMulAddH

```
// FPMatMulAddH()
// =====
// Half-precision matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

func FPMatMulAddH{N}(addend : bits(N), op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N == 128;
    let M : integer{} = 32;
    var result : bits(N);

    let isbfloat16 : boolean = FALSE;
    for i = 0 to 1 do
        for j = 0 to 1 do
            var sum : bits(M) = addend[(2*i + j)*M];
            var prod : array[[2]] of bits(M);
            for k = 0 to 1 do
                let elt1_a : bits(M DIV 2) = op1[(4*i + 2*k + 0)*(M DIV 2)];
                let elt1_b : bits(M DIV 2) = op1[(4*i + 2*k + 1)*(M DIV 2)];
                let elt2_a : bits(M DIV 2) = op2[(4*j + 2*k + 0)*(M DIV 2)];
                let elt2_b : bits(M DIV 2) = op2[(4*j + 2*k + 1)*(M DIV 2)];
                prod[[k]] = FPDot(elt1_a, elt1_b, elt2_a, elt2_b, fpcr, isbfloat16);
            end;
            sum = FPAdd{M}(sum, FPAdd{M}(prod[[0]], prod[[1]], fpcr), fpcr);
            result[(2*i + j)*M] = sum;
        end;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =====

func FPMax{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = TRUE;
    return FPMax{N}(op1, op2, fpcr, altfp, fpexc);
end;

// FPMax()
// =====

func FPMax{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, altfp : boolean) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return FPMax{N}(op1, op2, fpcr, altfp, fpexc);
end;

// FPMax()
// =====
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

func FPMax{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type,
    altfp : boolean, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type = fpcr_in;
    let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);

    if altfp && type1 == FType\_Zero && type2 == FType\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero{N}(sign2);
    elsif altfp && (type1 IN {FType\_SNaN, FType\_QNaN} || type2 IN {FType\_SNaN, FType\_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        return (if type2 == FType\_Zero then FPZero{N}(sign2) else op2);
    end;

    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        var fptype : FType;
        var sign : bit;
        var value : real;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        end;
        if fptype == FType\_Infinity then
            result = FPInfinity{N}(sign);
        elsif fptype == FType\_Zero then
            sign = sign1 AND sign2;           // Use most positive sign
            result = FPZero{N}(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            let rounding : FPRounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';
            end;

            result = FPRound{N}(value, fpcr, rounding, fpexc);
        end;
        if fpexc then FPProcessDenorms(type1, type2, N, fpcr); end;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaXNormal

```
// FPMaXNormal()
// =====

func FPMaXNormal{N}(sign : bit) => bits(N)
begin
  assert N IN {16,32,64};
  let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
  let F : integer{} = N - (E + 1);
  let exp : bits(E) = Ones{E-1}::'0';
  let frac : bits(F) = Ones{};
  return sign :: exp :: frac;
end;
```

Library pseudocode for shared/functions/float/fpmaxnum/FPMaXNum

```
// FPMaXNum()
// =====

func FPMaXNum{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  let fpexc : boolean = TRUE;
  return FPMaXNum{N}(op1, op2, fpcr, fpexc);
end;

// FPMaXNum()
// =====

func FPMaXNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
  assert N IN {16,32,64};
  var op1 : bits(N) = op1_in;
  var op2 : bits(N) = op2_in;
  let (type1,-,-) = FPUnpack{N}(op1, fpcr, fpexc);
  let (type2,-,-) = FPUnpack{N}(op2, fpcr, fpexc);

  let type1_nan : boolean = type1 IN {FPTType\_QNaN, FPTType\_SNaN};
  let type2_nan : boolean = type2 IN {FPTType\_QNaN, FPTType\_SNaN};
  let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';

  if !(altfp && type1_nan && type2_nan) then
    // Treat a single quiet-NaN as -Infinity.
    if type1 == FPTType\_QNaN && type2 != FPTType\_QNaN then
      op1 = FPInfinity{N}('1');
    elseif type1 != FPTType\_QNaN && type2 == FPTType\_QNaN then
      op2 = FPInfinity{N}('1');
    end;
  end;

  let altfmaxfmin : boolean = FALSE; // Restrict use of FMAX/FMIN NaN propagation rules
  let result : bits(N) = FPMaX{}(op1, op2, fpcr, altfmaxfmin, fpexc);

  return result;
end;
```

Library pseudocode for shared/functions/float/fpmerge/IsMerging

```
// IsMerging()
// =====
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

func IsMerging(fpcr : FPCR_Type) => boolean
begin
  let nep : bit = (if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' &&
    !IsFullA64Enabled() then '0' else fpcr.NEP);
  return IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && nep == '1';
end;
```

Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =====

func FPMin{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = TRUE;
    return FPMin{N}(op1, op2, fpcr, altfp, fpexc);
end;

// FPMin()
// =====

func FPMin{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, altfp : boolean) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return FPMin{N}(op1, op2, fpcr, altfp, fpexc);
end;

// FPMin()
// =====
// Compare two inputs and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

func FPMin{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type,
    altfp : boolean, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type = fpcr_in;
    let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);

    if altfp && type1 == FType\_Zero && type2 == FType\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero{N}(sign2);
    elsif altfp && (type1 IN {FType\_SNaN, FType\_QNaN} || type2 IN {FType\_SNaN, FType\_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        return (if type2 == FType\_Zero then FPZero{N}(sign2) else op2);
    end;

    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        var fptype : FType;
        var sign : bit;
        var value : real;
        var rounding : FPRounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        end;
        if fptype == FType\_Infinity then
            result = FPInfinity{N}(sign);
        elsif fptype == FType\_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero{N}(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';
            end;

            result = FPRound{N}(value, fpcr, rounding, fpexc);
        end;

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr); end;
    end;
    return result;
end;
```

Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// =====

func FPMinNum{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;
    return FPMinNum{N}(op1, op2, fpcr, fpexc);
end;

// FPMinNum()
// =====

func FPMinNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    var op1 : bits(N) = op1_in;
    var op2 : bits(N) = op2_in;
    let (type1,-,-) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,-,-) = FPUnpack{N}(op2, fpcr, fpexc);

    let type1_nan : boolean = type1 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    let type2_nan : boolean = type2 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &&
        fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPTYPE\_QNaN && type2 != FPTYPE\_QNaN then
            op1 = FPInfinity{N}('0');
        elseif type1 != FPTYPE\_QNaN && type2 == FPTYPE\_QNaN then
            op2 = FPInfinity{N}('0');
        end;
    end;

    let altfmaxfmin : boolean = FALSE; // Restrict use of FMAX/FMIN NaN propagation rules
    let result : bits(N) = FPMIn{N}(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
end;
```

Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =====

func FPMul{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    let (type1,sign1,value1) : (FPTYPE, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPTYPE, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean = (type1 == FPTYPE\_Infinity);
        let inf2 : boolean = (type2 == FPTYPE\_Infinity);
        let zero1 : boolean = (type1 == FPTYPE\_Zero);
        let zero2 : boolean = (type2 == FPTYPE\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN{N}(fpcr);
            FPPProcessException(FPEXC\_InvalidOp, fpcr);
        elseif inf1 || inf2 then
            result = FPInfinity{N}(sign1 XOR sign2);
        elseif zero1 || zero2 then
            result = FPZero{N}(sign1 XOR sign2);
        else
            result = FPRound{N}(value1*value2, fpcr);
        end;

        FPPProcessDenorms(type1, type2, N, fpcr);
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd

```
// FPMulAdd()
// =====

func FPMulAdd{N}(addend : bits(N), op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;          // Generate floating-point exceptions
    return FPMulAdd{N}(addend, op1, op2, fpcr, fpexc);
end;

// FPMulAdd()
// =====
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

func FPMulAdd{N}(addend : bits(N), op1 : bits(N), op2 : bits(N),
    fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};

    let (typeA,signA,valueA) : (FType, bit, real) = FPUnpack{N}(addend, fpcr, fpexc);
    let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr, fpexc);
    let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr, fpexc);
    let rounding : FPRounding = FPRoundingMode(fpcr);
    let inf1 : boolean = (type1 == FType Infinity); let zero1 : boolean = (type1 == FType Zero);
    let inf2 : boolean = (type2 == FType Infinity); let zero2 : boolean = (type2 == FType Zero);

    var (done,result) = FPProcessNaNs3{N}(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FType QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN{N}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        end;
    end;

    if !done then
        let infA : boolean = (typeA == FType Infinity);
        let zeroA : boolean = (typeA == FType Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        let signP : bit = sign1 XOR sign2;
        let infP : boolean = inf1 || inf2;
        let zeroP : boolean = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        let invalidop : boolean = ((inf1 && zero2) || (zero1 && inf2) ||
            (infA && infP && signA != signP));

        if invalidop then
            result = FPDefaultNaN{N}(fpcr);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity{N}('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity{N}('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero{N}(signA);

        // Otherwise calculate numerical result and round it.
        else
            let result_value : real = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding NEGINF then '1' else '0';
                result = FPZero{N}(result_sign);
            else
                result = FPRound{N}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if !invalidop && fpexc then
            FPProcessDenorms3(typeA, type1, type2, N, fpcr);
        end;
    end;
    return result;
end;
```


Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd_ZA

```
// FPMulAdd_ZA()
// =====
// Calculates addend + op1*op2 with a single rounding for SME ZA-targeting
// instructions.

func FPMulAdd_ZA{N}(addend : bits(N), op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    let fpexc : boolean = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';                // Generate default NaN values
    return FPMulAdd{N}(addend, op1, op2, fpcr, fpexc);
end;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

```
// FPMulAddH()
// =====
// Calculates addend + op1*op2.

func FPMulAddH(addend : bits(32), op1 : bits(16), op2 : bits(16), fpcr : FPCR_Type) => bits(32)
begin
    let fpexc : boolean = TRUE;          // Generate floating-point exceptions
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);
end;

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

func FPMulAddH(addend : bits(32), op1 : bits(16), op2 : bits(16),
               fpcr : FPCR_Type, fpexc : boolean) => bits(32)
begin
    let rounding : FPRounding = FPRoundingMode(fpcr);
    let (typeA, signA, valueA) = FPUnpack{32}(addend, fpcr, fpexc);
    let (type1, sign1, value1) = FPUnpack{16}(op1, fpcr, fpexc);
    let (type2, sign2, value2) = FPUnpack{16}(op2, fpcr, fpexc);
    let inf1 = (type1 == FPTYPE\_Infinity); let zero1 = (type1 == FPTYPE\_Zero);
    let inf2 = (type2 == FPTYPE\_Infinity); let zero2 = (type2 == FPTYPE\_Zero);

    var (done, result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPTYPE\_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr); end;
        end;
    end;

    if !done then
        let infA = (typeA == FPTYPE\_Infinity); let zeroA = (typeA == FPTYPE\_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        let signP = sign1 XOR sign2;
        let infP = inf1 || inf2;
        let zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        let invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN{32}(fpcr);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr); end;

        // Other cases involving infinities produce an infinity of the same sign.
        elseif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity{32}('0');
        elseif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity{32}('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elseif zeroA && zeroP && signA == signP then
            result = FPZero{32}(signA);

        // Otherwise calculate numerical result and round it.
        else
            let result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                let result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero{32}(result_sign);
            else
                result = FPRound{32}(result_value, fpcr, rounding, fpexc);
            end;
        end;

        if !invalidop && fpexc then
            FPProcessDenorm(typeA, 32, fpcr);
        end;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH_ZA

```
// FPMulAddH_ZA()
// =====
// Calculates addend + op1*op2 for SME2 ZA-targeting instructions.

func FPMulAddH_ZA(addend : bits(32), op1 : bits(16), op2 : bits(16),
    fpcr_in : FPCR_Type) => bits(32)
begin
    var fpcr : FPCR_Type = fpcr_in;
    let fpexc : boolean = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1'; // Generate default NaN values
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);
end;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =====

func FPProcessNaNs3H(type1 : FPTYPE, type2 : FPTYPE, type3 : FPTYPE,
    op1 : bits(32), op2 : bits(16), op3 : bits(16),
    fpcr : FPCR_Type, fpexc : boolean) => (boolean, bits(32))
begin
    var result : bits(32);
    var type_nan : FPTYPE;
    // When TRUE, use alternative NaN propagation rules.
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let op1_nan : boolean = type1 IN {FPTYPE\_SNaN, FPTYPE\_QNaN};
    let op2_nan : boolean = type2 IN {FPTYPE\_SNaN, FPTYPE\_QNaN};
    let op3_nan : boolean = type3 IN {FPTYPE\_SNaN, FPTYPE\_QNaN};
    if altfp then
        if (type1 == FPTYPE\_SNaN || type2 == FPTYPE\_SNaN || type3 == FPTYPE\_SNaN) then
            type_nan = FPTYPE\_SNaN;
        else
            type_nan = FPTYPE\_QNaN;
        end;
    end;

    var done : boolean;
    if altfp && op1_nan && op2_nan && op3_nan then // [n] register NaN selected
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type_nan, op2, fpcr, fpexc));
    elsif altfp && op2_nan && (op1_nan || op3_nan) then // [n] register NaN selected
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type_nan, op2, fpcr, fpexc));
    elsif altfp && op3_nan && op1_nan then // [m] register NaN selected
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type_nan, op3, fpcr, fpexc));
    elsif type1 == FPTYPE\_SNaN then
        done = TRUE; result = FPProcessNaN{32}(type1, op1, fpcr, fpexc);
    elsif type2 == FPTYPE\_SNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type2, op2, fpcr, fpexc));
    elsif type3 == FPTYPE\_SNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type3, op3, fpcr, fpexc));
    elsif type1 == FPTYPE\_QNaN then
        done = TRUE; result = FPProcessNaN{32}(type1, op1, fpcr, fpexc);
    elsif type2 == FPTYPE\_QNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type2, op2, fpcr, fpexc));
    elsif type3 == FPTYPE\_QNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPProcessNaN{16}(type3, op3, fpcr, fpexc));
    else
        done = FALSE; result = Zeros{32}; // 'Don't care' result
    end;
    return (done, result);
end;
```

Library pseudocode for shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// =====

func FPMulX{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  assert N IN {16,32,64};
  var result : bits(N);
  var done : boolean;
  let (type1,sign1,value1) : (FType, bit, real) = FUnpack{N}(op1, fpcr);
  let (type2,sign2,value2) : (FType, bit, real) = FUnpack{N}(op2, fpcr);

  (done,result) = FProcessNaNs{N}(type1, type2, op1, op2, fpcr);
  if !done then
    let inf1 : boolean = (type1 == FType_Infinity);
    let inf2 : boolean = (type2 == FType_Infinity);
    let zero1 : boolean = (type1 == FType_Zero);
    let zero2 : boolean = (type2 == FType_Zero);

    if (inf1 && zero2) || (zero1 && inf2) then
      result = FPTwo{N}(sign1 XOR sign2);
    elsif inf1 || inf2 then
      result = FPinfinity{N}(sign1 XOR sign2);
    elsif zero1 || zero2 then
      result = FPZero{N}(sign1 XOR sign2);
    else
      result = FPRound{N}(value1*value2, fpcr);
    end;

    FProcessDenorms(type1, type2, N, fpcr);

  end;
  return result;
end;
```

Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =====

func FPNeg{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  assert N IN {16,32,64};
  if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
    if fpcr.AH == '1' then
      let (fptype, -, -) = FUnpack{N}(op, fpcr, FALSE);
      if fptype IN {FType_SNaN, FType_QNaN} then
        return op; // When fpcr.AH=1, sign of NaN has no consequence
      end;
    end;
  end;
  return NOT(op[N-1]) :: op[N-2:0];
end;
```

Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// =====

func FPOnePointFive{N}(sign : bit) => bits(N)
begin
  assert N IN {16,32,64};
  let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
  let F : integer{} = N - (E + 1);
  let exp : bits(E) = '0'::Ones{E-1};
  let frac : bits(F) = '1'::Zeros{F-1};
  let result : bits((F + E) + 1) = sign :: exp :: frac;

  return result;
end;
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPPProcessDenorm

```
// FPPProcessDenorm()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

func FPPProcessDenorm(fptype : FPTYPE, N : integer, fpcr : FPCR_Type)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPTYPE\_Denormal then
        FPPProcessException(FPExc\_InputDenorm, fpcr);
    end;
end;
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPPProcessDenorms

```
// FPPProcessDenorms()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

func FPPProcessDenorms(type1 : FPTYPE, type2 : FPTYPE, N : integer, fpcr : FPCR_Type)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPTYPE\_Denormal || type2 == FPTYPE\_Denormal) then
        FPPProcessException(FPExc\_InputDenorm, fpcr);
    end;
end;
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPPProcessDenorms3

```
// FPPProcessDenorms3()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

func FPPProcessDenorms3(type1 : FPTYPE, type2 : FPTYPE, type3 : FPTYPE, N : integer,
                        fpcr : FPCR_Type)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPTYPE\_Denormal || type2 == FPTYPE\_Denormal ||
                          type3 == FPTYPE\_Denormal) then
        FPPProcessException(FPExc\_InputDenorm, fpcr);
    end;
end;
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPPProcessDenorms4

```
// FPPProcessDenorms4()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

func FPPProcessDenorms4(type1 : FPTYPE, type2 : FPTYPE, type3 : FPTYPE, type4 : FPTYPE,
                        N : integer, fpcr : FPCR_Type)
begin
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPTYPE\_Denormal || type2 == FPTYPE\_Denormal ||
                          type3 == FPTYPE\_Denormal || type4 == FPTYPE\_Denormal) then
        FPPProcessException(FPExc\_InputDenorm, fpcr);
    end;
end;
```

Library pseudocode for shared/functions/float/fpprocessexception/FPPProcessException

```
// FPPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

func FPPProcessException(except : FPExc, fpcr : FPCR_Type)
begin
    var cumul : integer;
    // Determine the cumulative exception bit number
    case except of
        when FPExc_InvalidOp =>    cumul = 0;
        when FPExc_DivideByZero => cumul = 1;
        when FPExc_Overflow  =>    cumul = 2;
        when FPExc_Underflow =>    cumul = 3;
        when FPExc_Inexact   =>    cumul = 4;
        when FPExc_InputDenorm =>   cumul = 7;
    end;
    let enable : integer = cumul + 8;
    if (fpcr[enable] == '1' && (!IsFeatureImplemented(FEAT_SME) || PSTATE.SM == '0' ||
        IsFullA64Enabled())) then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
        // and if so then how exceptions and in what order that they may be
        // accumulated before calling FPTrappedException().
        var accumulated_exceptions : bits(8) = GetAccumulatedFPEExceptions();
        accumulated_exceptions[cumul] = '1';
        if ImpDefBool("Support trapping of floating-point exceptions") then
            if UsingAArch32() then
                AArch32_FPTrappedException(accumulated_exceptions);
            else
                let is_ase : boolean = IsASEInstruction();
                AArch64_FPTrappedException(is_ase, accumulated_exceptions);
            end;
        else
            // The exceptions generated by this instruction are accumulated by the PE and
            // FPTrappedException is called later during its execution, before the next
            // instruction is executed. This field is cleared at the start of each FP instruction.
            SetAccumulatedFPEExceptions(accumulated_exceptions);
        end;
    elseif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR()[cumul] = '1';
    else
        // Set the cumulative exception bit
        FPSR()[cumul] = '1';
    end;
    return;
end;
```

Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// =====

func FPProcessNaN{N}(fptype : FPTYPE, op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    let fpexc : boolean = TRUE;    // Generate floating-point exceptions
    return FPProcessNaN{N}(fptype, op, fpcr, fpexc);
end;

// FPProcessNaN()
// =====
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

func FPProcessNaN{N}(fptype : FPTYPE, op : bits(N), fpcr : FPCR_Type, fpexc : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    assert fptype IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    var topfrac : integer;

    case N of
        when 16 => topfrac = 9;
        when 32 => topfrac = 22;
        when 64 => topfrac = 51;
    end;

    var result : bits(N) = op;
    if fptype == FPTYPE\_SNaN then
        result[topfrac] = '1';
        if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr); end;
    end;
    if fpcr.DN == '1' then // DefaultNaN requested
        result = FPDefaultNaN{N}(fpcr);
    end;
    return result;
end;
```

Library pseudocode for shared/functions/float/fpprocessnans/FPPProcessNaNs

```
// FPPProcessNaNs()
// =====

func FPPProcessNaNs{N}(type1 : FPTYPE, type2 : FPTYPE, op1 : bits(N),
                        op2 : bits(N), fpcr : FPCR\_Type) => (boolean, bits(N))
begin
    let fpexc : boolean = TRUE;      // Generate floating-point exceptions
    return FPPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpexc);
end;

// FPPProcessNaNs()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behavior for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

func FPPProcessNaNs{N}(type1 : FPTYPE, type2 : FPTYPE, op1 : bits(N), op2 : bits(N),
                        fpcr : FPCR\_Type, fpexc : boolean) => (boolean, bits(N))
begin
    assert N IN {16,32,64};
    var done : boolean;
    var result : bits(N);
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let op1_nan : boolean = type1 IN {FPTYPE\_SNaN, FPTYPE\_QNaN};
    let op2_nan : boolean = type2 IN {FPTYPE\_SNaN, FPTYPE\_QNaN};
    let any_snan : boolean = type1 == FPTYPE\_SNaN || type2 == FPTYPE\_SNaN;
    let type_nan : FPTYPE = if any_snan then FPTYPE\_SNaN else FPTYPE\_QNaN;

    if altfp && op1_nan && op2_nan then
        // [n] register NaN selected
        done = TRUE; result = FPPProcessNaN{N}(type_nan, op1, fpcr, fpexc);
    elsif type1 == FPTYPE\_SNaN then
        done = TRUE; result = FPPProcessNaN{N}(type1, op1, fpcr, fpexc);
    elsif type2 == FPTYPE\_SNaN then
        done = TRUE; result = FPPProcessNaN{N}(type2, op2, fpcr, fpexc);
    elsif type1 == FPTYPE\_QNaN then
        done = TRUE; result = FPPProcessNaN{N}(type1, op1, fpcr, fpexc);
    elsif type2 == FPTYPE\_QNaN then
        done = TRUE; result = FPPProcessNaN{N}(type2, op2, fpcr, fpexc);
    else
        done = FALSE; result = Zeros{N}; // 'Don't care' result
    end;
    return (done, result);
end;
```


Library pseudocode for shared/functions/float/fpprocessnans3/FPPProcessNaNs3

```
// FPPProcessNaNs3()
// =====

func FPPProcessNaNs3{N}(type1 : FType, type2 : FType, type3 : FType,
    op1 : bits(N), op2 : bits(N), op3 : bits(N),
    fpcr : FPCR_Type) => (boolean, bits(N))

begin
    let fpexc : boolean = TRUE;    // Generate floating-point exceptions
    return FPPProcessNaNs3{N}(type1, type2, type3, op1, op2, op3, fpcr, fpexc);
end;

// FPPProcessNaNs3()
// =====
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

func FPPProcessNaNs3{N}(type1 : FType, type2 : FType, type3 : FType,
    op1 : bits(N), op2 : bits(N), op3 : bits(N),
    fpcr : FPCR_Type, fpexc : boolean) => (boolean, bits(N))

begin
    assert N IN {16,32,64};
    var result : bits(N);
    let op1_nan : boolean = type1 IN {FType\_SNaN, FType\_QNaN};
    let op2_nan : boolean = type2 IN {FType\_SNaN, FType\_QNaN};
    let op3_nan : boolean = type3 IN {FType\_SNaN, FType\_QNaN};

    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    var type_nan : FType;
    if altfp then
        if type1 == FType\_SNaN || type2 == FType\_SNaN || type3 == FType\_SNaN then
            type_nan = FType\_SNaN;
        else
            type_nan = FType\_QNaN;
        end;
    end;

    var done : boolean;
    if altfp && op1_nan && op2_nan && op3_nan then
        // [n] register NaN selected
        done = TRUE; result = FPPProcessNaN{N}(type_nan, op2, fpcr, fpexc);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        // [n] register NaN selected
        done = TRUE; result = FPPProcessNaN{N}(type_nan, op2, fpcr, fpexc);
    elsif altfp && op3_nan && op1_nan then
        // [m] register NaN selected
        done = TRUE; result = FPPProcessNaN{N}(type_nan, op3, fpcr, fpexc);
    elsif type1 == FType\_SNaN then
        done = TRUE; result = FPPProcessNaN{N}(type1, op1, fpcr, fpexc);
    elsif type2 == FType\_SNaN then
        done = TRUE; result = FPPProcessNaN{N}(type2, op2, fpcr, fpexc);
    elsif type3 == FType\_SNaN then
        done = TRUE; result = FPPProcessNaN{N}(type3, op3, fpcr, fpexc);
    elsif type1 == FType\_QNaN then
        done = TRUE; result = FPPProcessNaN{N}(type1, op1, fpcr, fpexc);
    elsif type2 == FType\_QNaN then
        done = TRUE; result = FPPProcessNaN{N}(type2, op2, fpcr, fpexc);
    elsif type3 == FType\_QNaN then
        done = TRUE; result = FPPProcessNaN{N}(type3, op3, fpcr, fpexc);
    else
        done = FALSE; result = Zeros{N}; // 'Don't care' result
    end;
    return (done, result);
end;
```

Library pseudocode for shared/functions/float/fpprocessnans4/FPPProcessNaNs4

```
// FPPProcessNaNs4()
// =====
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits.
// Status information is updated directly in the FPSR where appropriate.
// The 'fpexc' controls the generation of floating-point exceptions.

func FPPProcessNaNs4(type1 : FPType, type2 : FPType, type3 : FPType, type4 : FPType,
    op1 : bits(16), op2 : bits(16), op3 : bits(16),
    op4 : bits(16), fpcr : FPCR_Type, fpexc : boolean) => (boolean, bits(32))
begin
    var result : bits(32);
    var done : boolean;
    // The FPCR.AH control does not affect these checks
    if type1 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPPProcessNaN{16}(type1, op1, fpcr, fpexc));
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPPProcessNaN{16}(type2, op2, fpcr, fpexc));
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPPProcessNaN{16}(type3, op3, fpcr, fpexc));
    elsif type4 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPPProcessNaN{16}(type4, op4, fpcr, fpexc));
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPPProcessNaN{16}(type1, op1, fpcr, fpexc));
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPPProcessNaN{16}(type2, op2, fpcr, fpexc));
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPPProcessNaN{16}(type3, op3, fpcr, fpexc));
    elsif type4 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN{32, 16}(FPPProcessNaN{16}(type4, op4, fpcr, fpexc));
    else
        done = FALSE; result = Zeros{32}; // 'Don't care' result
    end;

    return (done, result);
end;
```



```

// FPREcipEstimate()
// =====

func FPREcipEstimate(N)(operand : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var fpcr : FPCR_Type = fpcr_in;
    var result : bits(N);
    var overflow_to_inf : boolean;
    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = !altfp;
    if altfp then fpcr.[FIZ,FZ] = '11'; end;
    if altfp then fpcr.RMode = '00'; end;

    let (fptype,sign,value) : (FPTType, bit, real) = FPUnpack{N}(operand, fpcr, fpexc);

    let rounding : FPRounding = FPRoundingMode(fpcr);
    if fptype == FPTType_SNaN || fptype == FPTType_QNaN then
        result = FPProcessNaN{N}(fptype, operand, fpcr, fpexc);
    elseif fptype == FPTType_Infinity then
        result = FPZero{N}(sign);
    elseif fptype == FPTType_Zero then
        result = FPInfinity{N}(sign);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr); end;
    elseif (
        (N == 16 && Abs(value) < 2.0^-16) ||
        (N == 32 && Abs(value) < 2.0^-128) ||
        (N == 64 && Abs(value) < 2.0^-1024)
    ) then
        case rounding of
            when FPRounding_TIEEVEN =>
                overflow_to_inf = TRUE;
            when FPRounding_POSINF =>
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF =>
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO =>
                overflow_to_inf = FALSE;
        end;
        result = if overflow_to_inf then FPInfinity{N}(sign) else FPMaxNormal{N}(sign);
    if fpexc then
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    end;
    elseif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
        && (
            (N == 16 && Abs(value) >= 2.0^14) ||
            (N == 32 && Abs(value) >= 2.0^126) ||
            (N == 64 && Abs(value) >= 2.0^1022)
        ) then
        // Result flushed to zero of correct sign
        result = FPZero{N}(sign);

        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSR().UFC = '1';
        else
            if fpexc then FPSR().UFC = '1'; end;
        end;
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        var fraction : bits(52);
        var exp : integer;
        case N of
            when 16 =>
                fraction = operand[9:0] :: Zeros{42};
                exp = UInt(operand[14:10]);
            when 32 =>
                fraction = operand[22:0] :: Zeros{29};
                exp = UInt(operand[30:23]);
            when 64 =>
                fraction = operand[51:0];
                exp = UInt(operand[62:52]);
        end;

        if exp == 0 then
            if fraction[51] == '0' then
                exp = -1;
                fraction = fraction[49:0]::'00';
            else
                fraction = fraction[50:0]::'0';
            end;
        end;
    end;
end;

```

```

var scaled : integer;
let increasedprecision : boolean = N==32 && IsFeatureImplemented(FEAT_RPRES) && altfp;

if !increasedprecision then
    scaled = UInt('1'::fraction[51:44]);
else
    scaled = UInt('1'::fraction[51:41]);
end;

var result_exp : integer;
case N of
    when 16 => result_exp = 29 - exp;    // In range 29-30 = -1 to 29+1 = 30
    when 32 => result_exp = 253 - exp;   // In range 253-254 = -1 to 253+1 = 254
    when 64 => result_exp = 2045 - exp;  // In range 2045-2046 = -1 to 2045+1 = 2046
end;

// Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
// fixed-point number in range [0.5 .. 1.0].
let estimate : integer = RecipEstimate(scaled, increasedprecision);

// Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit,
// high-order bits from estimate, and exponent calculated above.
if !increasedprecision then
    fraction = estimate[7:0] :: Zeros{44};
else
    fraction = estimate[11:0] :: Zeros{40};
end;

if result_exp == 0 then
    fraction = '1' :: fraction[51:1];
elseif result_exp == -1 then
    fraction = '01' :: fraction[51:2];
    result_exp = 0;
end;

case N of
    when 16 => result = sign :: result_exp[N-12:0] :: fraction[51:42];
    when 32 => result = sign :: result_exp[N-25:0] :: fraction[51:29];
    when 64 => result = sign :: result_exp[N-54:0] :: fraction[51:0];
end;
end;

return result;
end;

```

Library pseudocode for shared/functions/float/fpreciestimate/RecipEstimate

```

// RecipEstimate()
// =====
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

func RecipEstimate(a_in : integer, increasedprecision : boolean) => integer
begin
    var a : integer = a_in;
    var r : integer;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1;                      // Round to nearest
        let b : integer = (2 ^ 19) DIVRM a;
        r = (b+1) DIVRM 2;              // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1;                      // Round to nearest
        let b : integer = (2 ^ 26) DIVRM a;
        r = (b+1) DIVRM 2;              // Round to nearest
        assert 4096 <= r && r < 8192;
    end;
    return r;
end;

```

Library pseudocode for shared/functions/float/fprecpx/FPrecpX

```
// FPrecpX()
// =====

func FPrecpX{N}(op : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var fpcr : FPCR_Type = fpcr_in;
    let isbfloat16 : boolean = FALSE;
    let (F, -) = FPBits(N, isbfloat16);
    let E = (N - F) - 1;
    var result : bits(N);
    var exp : bits(E);
    var max_exp : bits(E);
    let frac : bits(F) = Zeros{};

    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    let fpexc : boolean = !altfp;           // Generate no floating-point exceptions
    if altfp then fpcr.[FIZ,FZ] = '11'; end; // Flush denormal input and output to zero
    let (fptype,sign,value) = FPUntpack{N}(op, fpcr, fpexc);
    exp = op[F+:E];
    max_exp = Ones{E} - 1;

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN{N}(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then                // Zero and denormals
            result = ZeroExtend{N}(sign::max_exp::frac);
        else                               // Infinities and normals
            result = ZeroExtend{N}(sign::NOT(exp)::frac);
        end;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fpround/FPRound

```
// FPRound()
// =====
// Generic conversion from precise, unbounded real data type to IEEE format.

func FPRound{N}(op : real, fpcr : FPCR_Type) => bits(N)
begin
    return FPRound{N}(op, fpcr, FPRoundingMode(fpcr));
end;

// FPRound()
// =====
// For directed FP conversion, includes an explicit 'rounding' argument.

func FPRound{N}(op : real, fpcr_in : FPCR_Type, rounding : FPRounding) => bits(N)
begin
    let fpexc : boolean = TRUE;           // Generate floating-point exceptions
    return FPRound{N}(op, fpcr_in, rounding, fpexc);
end;

// FPRound()
// =====
// For AltFP, includes an explicit FPEXC argument to disable exception
// generation and switches off Arm alternate half-precision mode.

func FPRound{N}(op : real, fpcr_in : FPCR_Type, rounding : FPRounding, fpexc : boolean) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    let isbfloat16 : boolean = FALSE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc);
end;
```



```

// FPRoundBase()
// =====
// For BFloat16, includes an explicit 'isbfloat16' argument.

func FPRoundBase{N}(op : real, fpcr : FPCR_Type, rounding : FPRounding,
    isbfloat16 : boolean) => bits(N)
begin
    let fpexc : boolean = TRUE;    // Generate floating-point exceptions
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc);
end;

// FPRoundBase()
// =====
// For FP8 multiply-accumulate, dot product, and outer product instructions, includes
// an explicit saturation overflow argument.

func FPRoundBase{N}(op : real, fpcr : FPCR_Type, rounding : FPRounding, isbfloat16 : boolean,
    fpexc : boolean) => bits(N)
begin
    let satoflo : boolean = FALSE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc, satoflo);
end;

// FPRoundBase()
// =====
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate. The 'satoflo' argument
// controls whether overflow generates Infinity or MaxNorm for 8-bit floating-point
// data processing instructions.

func FPRoundBase{N}(op : real, fpcr : FPCR_Type, rounding : FPRounding, isbfloat16 : boolean,
    fpexc : boolean, satoflo : boolean) => bits(N)
begin
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding\_TIEAWAY;
    var result : bits(N);

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    let (F, minimum_exp) = FPBits(N, isbfloat16);
    let zeros = if N == 32 && isbfloat16 then 16 else 0;
    let E = N - (F + 1 + zeros);
    // Split value into sign, unrounded mantissa and exponent.
    var sign : bit;
    var exponent : integer;
    var mantissa : real;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    end;
    (mantissa, exponent) = NormalizeReal(mantissa);

    // When TRUE, detection of underflow occurs after rounding and the test for a
    // denormalized number for single and double precision values occurs after rounding.
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';

    // Deal with flush-to-zero before rounding if FPCR.AH != '1'.
    if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
        exponent < minimum_exp) then
        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR().UFC = '1';
        else
            if fpexc then FPSR().UFC = '1'; end;
        end;
        return FPZero{N}(sign);
    end;

    var biased_exp_unconstrained : integer = (exponent - minimum_exp) + 1;
    var int_mant_unconstrained : integer = RoundDown(mantissa * 2.0^F);
    let error_unconstrained : real = mantissa * 2.0^F - Real(int_mant_unconstrained);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    var biased_exp : integer = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent); end;

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    var int_mant : integer = RoundDown(mantissa * 2.0^F);
    var error : real = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or

```



```

// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
let trapped_UF : boolean = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());
if !altfp && biased_exp == 0 && (error != 0.0 || trapped_UF) then
    if fpexc then FProcessException\(FPExc Underflow, fpcr\); end;
end;

// Round result according to rounding mode.
var round_up_unconstrained : boolean;
var round_up : boolean;
var overflow_to_inf : boolean;
if altfp then

    case rounding of
        when FPRounding TIEEVEN =>
            round_up_unconstrained = (error_unconstrained > 0.5 ||
                (error_unconstrained == 0.5 && int_mant_unconstrained[0] == '1'));
            round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));
            overflow_to_inf = !satoflo;
        when FPRounding POSINF =>
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding NEGINF =>
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding ZERO, FPRounding ODD =>
            round_up_unconstrained = FALSE;
            round_up = FALSE;
            overflow_to_inf = FALSE;
    end;

    if round_up_unconstrained then
        int_mant_unconstrained = int_mant_unconstrained + 1;
        if int_mant_unconstrained == 2^(F+1) then // Rounded up to next exponent
            biased_exp_unconstrained = biased_exp_unconstrained + 1;
            int_mant_unconstrained = int_mant_unconstrained DIV 2;
        end;
    end;

    // Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
    if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
        // the result of unconstrained rounding is less than the minimum normalized number
        if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then // Flush-to-zero
            if fpexc then
                FPSR().UFC = '1';
                FProcessException\(FPExc Inexact, fpcr\);
            end;
            return FPZero{N}(sign);
        elsif error != 0.0 || trapped_UF then
            if fpexc then FProcessException\(FPExc Underflow, fpcr\); end;
        end;
    end;
else // altfp == FALSE
    case rounding of
        when FPRounding TIEEVEN =>
            round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));
            overflow_to_inf = !satoflo;
        when FPRounding POSINF =>
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding NEGINF =>
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding ZERO, FPRounding ODD =>
            round_up = FALSE;
            overflow_to_inf = FALSE;
    end;
end;

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then // Rounded up from denormalized to normalized
        biased_exp = 1;
    end;
    if int_mant == 2^(F+1) then // Rounded up to next exponent
        biased_exp = biased_exp + 1;
        int_mant = int_mant DIV 2;
    end;
end;

// Handle rounding to odd
if error != 0.0 && rounding == FPRounding ODD && int_mant[0] == '0' then
    int_mant = int_mant + 1;
end;

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then

```

```

        result = if overflow_to_inf then FPInfinity{N}(sign) else FPMMaxNormal{N}(sign);
        if fpexc then FPPProcessException(FPExc\_Overflow, fpcr); end;
        error = 1.0; // Ensure that an Inexact exception occurs
    else
        result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0] :: Zeros{N-(E+F+1)};
    end;
else
    // Alternative half precision
    if biased_exp >= 2^E then
        result = sign :: Ones{N-1};
        if fpexc then FPPProcessException(FPExc\_InvalidOp, fpcr); end;
        error = 0.0; // Ensure that an Inexact exception does not occur
    else
        result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0] :: Zeros{N-(E+F+1)};
    end;
end;

// Deal with Inexact exception.
if error != 0.0 then
    if fpexc then FPPProcessException(FPExc\_Inexact, fpcr); end;
end;

return result;
end;
```

Library pseudocode for shared/functions/float/fpround/FPRoundCV

```

// FPRoundCV()
// =====
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

func FPRoundCV{N}(op : real, fpcr_in : FPCR_Type, rounding : FPRounding) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.FZ16 = '0';
    let fpexc : boolean = TRUE; // Generate floating-point exceptions
    let isbfloat16 : boolean = FALSE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc);
end;
```

Library pseudocode for shared/functions/float/fpround/FPRound_FP8

```

// FPRound_FP8()
// =====
// Used by FP8 multiply-accumulate, dot product, and outer product instructions
// which observe FPMR.OSM.

func FPRound_FP8{N}(op : real, fpcr_in : FPCR_Type, rounding : FPRounding,
    satoflo : boolean) => bits(N)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    let fpexc : boolean = FALSE;
    let isbfloat16 : boolean = FALSE;
    return FPRoundBase{N}(op, fpcr, rounding, isbfloat16, fpexc, satoflo);
end;
```

Library pseudocode for shared/functions/float/fprounding/FPRounding

```

// FPRounding
// =====
// The conversion and rounding functions take an explicit
// rounding mode enumeration instead of booleans or FPCR values.

type FPRounding of enumeration {FPRounding_TIEEVEN, FPRounding_POSINF,
    FPRounding_NEGINF, FPRounding_ZERO,
    FPRounding_TIEAWAY, FPRounding_ODD};
```

Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

```

// FPRoundingMode()
// =====
// Return the current floating-point rounding mode.

func FPRoundingMode(fpcr : FPCR_Type) => FPRounding
begin
    return FPDecodeRounding(fpcr.RMode);
end;
```

Library pseudocode for shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// =====

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

func FPRoundInt{N}(op : bits(N), fpcr : FPCR_Type, rounding : FPRounding,
    exact : boolean) => bits(N)

begin
    assert rounding != FPRounding\_ODD;
    assert N IN {16,32,64};

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    let (fptype,sign,value) = FPUnpack{N}(op, fpcr, fpexc);

    var result : bits(N);
    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        result = FPProcessNaN{N}(fptype, op, fpcr);
    elsif fptype == FPTYPE\_Infinity then
        result = FPInfinity{N}(sign);
    elsif fptype == FPTYPE\_Zero then
        result = FPZero{N}(sign);
    else
        // Extract integer component.
        var int_result : integer = RoundDown(value);
        let error : real = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        var round_up : boolean;
        case rounding of
            when FPRounding\_TIEEVEN =>
                round_up = (error > 0.5 || (error == 0.5 && int_result[0] == '1'));
            when FPRounding\_POSINF =>
                round_up = (error != 0.0);
            when FPRounding\_NEGINF =>
                round_up = FALSE;
            when FPRounding\_ZERO =>
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding\_TIEAWAY =>
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
        end;

        if round_up then int_result = int_result + 1; end;

        // Convert integer value into an equivalent real value.
        let real_result : real = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero{N}(sign);
        else
            result = FPRound{N}(real_result, fpcr, FPRounding\_ZERO);
        end;

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc\_Inexact, fpcr);
        end;
    end;

    return result;
end;
```



```

// FPRoundIntN()
// =====

func FPRoundIntN{N}(op : bits(N), fpcr : FPCR_Type,
                    rounding : FPRounding, intsize : integer) => bits(N)
begin
    assert rounding != FPRounding\_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    var exp : integer;
    var result : bits(N);
    var round_up : boolean;
    let E : integer{} = (if N == 32 then 8 else 11);
    let F : integer{} = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    let altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpxc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    let (fptype,sign,value) = FPUnpack{N}(op, fpcr, fpxc);

    if fptype IN {FPTYPE\_SNaN, FPTYPE\_QNaN, FPTYPE\_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1'::exp[(E-1):0]::Zeros{F};
        else
            exp = 1022+intsize;
            result = '1'::exp[(E-1):0]::Zeros{F};
        end;
        FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif fptype == FPTYPE\_Zero then
        result = FPZero{N}(sign);
    else
        // Extract integer component.
        var int_result = RoundDown(value);
        var error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        case rounding of
            when FPRounding\_TIEEVEN =>
                round_up = error > 0.5 || (error == 0.5 && int_result[0] == '1');
            when FPRounding\_POSINF =>
                round_up = error != 0.0;
            when FPRounding\_NEGINF =>
                round_up = FALSE;
            when FPRounding\_ZERO =>
                round_up = error != 0.0 && int_result < 0;
            when FPRounding\_TIEAWAY =>
                round_up = error > 0.5 || (error == 0.5 && int_result >= 0);
        end;

        if round_up then int_result = int_result + 1; end;
        let overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

        if overflow then
            if N == 32 then
                exp = 126 + intsize;
                result = '1'::exp[(E-1):0]::Zeros{F};
            else
                exp = 1022 + intsize;
                result = '1'::exp[(E-1):0]::Zeros{F};
            end;
            FPProcessException(FPExc\_InvalidOp, fpcr);
            // This case shouldn't set Inexact.
            error = 0.0;
        else
            // Convert integer value into an equivalent real value.
            let real_result : real = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero{N}(sign);
            else
                result = FPRound{N}(real_result, fpcr, FPRounding\_ZERO);
            end;
        end;

        // Generate inexact exceptions.
        if error != 0.0 then
            FPProcessException(FPExc\_Inexact, fpcr);
        end;
    end;

    return result;
end;

```



```

// FPRSqrtEstimate()
// =====

func FPRSqrtEstimate(N)(operand : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    var fpcr : FPCR_Type = fpcr_in;

    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions and flush denormal input to zero.
    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc : boolean = !altfp;
    if altfp then fpcr.[FIZ,FZ] = '11'; end;

    let (fptype,sign,value) : (FType, bit, real) = FPUnpack{N}(operand, fpcr, fpexc);

    var result : bits(N);
    if fptype == FType_SNaN || fptype == FType_QNaN then
        result = FPProcessNaN{N}(fptype, operand, fpcr, fpexc);
    elseif fptype == FType_Zero then
        result = FPInfinity{N}(sign);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr); end;
    elseif sign == '1' then
        result = FPDefaultNaN{N}(fpcr);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr); end;
    elseif fptype == FType_Infinity then
        result = FPZero{N}('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        var fraction : bits(52);
        var exp : integer;
        case N of
            when 16 =>
                fraction = operand[9:0] :: Zeros{42};
                exp = UInt(operand[14:10]);
            when 32 =>
                fraction = operand[22:0] :: Zeros{29};
                exp = UInt(operand[30:23]);
            when 64 =>
                fraction = operand[51:0];
                exp = UInt(operand[62:52]);
        end;

        if exp == 0 then
            while fraction[51] == '0' looplimit 51 do
                fraction = fraction[50:0] :: '0';
                exp = exp - 1;
            end;
            fraction = fraction[50:0] :: '0';
        end;

        var scaled : integer;
        let increasedprecision : boolean = N==32 && IsFeatureImplemented(FEAT_RPRES) && altfp;

        if !increasedprecision then
            if exp[0] == '0' then
                scaled = UInt('1'::fraction[51:44]);
            else
                scaled = UInt('01'::fraction[51:45]);
            end;
        else
            if exp[0] == '0' then
                scaled = UInt('1'::fraction[51:41]);
            else
                scaled = UInt('01'::fraction[51:42]);
            end;
        end;

        var result_exp : integer;
        case N of
            when 16 => result_exp = ( 44 - exp) DIVRM 2;
            when 32 => result_exp = ( 380 - exp) DIVRM 2;
            when 64 => result_exp = (3068 - exp) DIVRM 2;
        end;

        let estimate : integer = RecipSqrtEstimate(scaled, increasedprecision);

        // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
        // fixed-point result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 => result = '0' :: result_exp[N-12:0] :: estimate[7:0]::Zeros{2};
            when 32 =>
                if !increasedprecision then

```

```

        result = '0' :: result_exp[N-25:0] :: estimate[7:0]::Zeros{15};
    else
        result = '0' :: result_exp[N-25:0] :: estimate[11:0]::Zeros{11};
    end;
    when 64 => result = '0' :: result_exp[N-54:0] :: estimate[7:0]::Zeros{44};
end;
end;

return result;
end;

```

Library pseudocode for shared/functions/float/fprsqtestimate/RecipSqrtEstimate

```

// RecipSqrtEstimate()
// =====
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

func RecipSqrtEstimate(a_in : integer, increasedprecision : boolean) => integer
begin
    var a : integer = a_in;
    var r : integer;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then
            // a in [128, 255], represents a value in [0.25, 0.5)
            // promote to 9-bit range in units of 1/512
            a = a*2+1;
        else
            // a in [256, 511], represents a value in [0.5, 1.0)
            // Discard bottom bit
            a = (a >> 1) << 1;
            // round up to nearest 1/256 and convert
            a = (a+1)*2;
        end;
        var b : integer = 512;
        while a*(b+1)*(b+1) < 2^28 looplimit 510 do
            b = b+1;
        end;
        // b = largest b such that b < 2^14 / sqrt(a)
        r = (b+1) DIVRM 2;
        // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        if a < 2048 then
            // a in [1024, 2047], represents a value in [0.25, 0.5)
            // promote to 13-bit range in units of 1/8192
            a = a*2 + 1;
        else
            // a in [2048, 4095], represents a value in [0.5, 1.0)
            // Discard bottom bit
            a = (a >> 1) << 1;
            // round up to nearest 1/4096 and convert
            a = (a+1)*2;
        end;
        var b : integer = 8192;
        while a*(b+1)*(b+1) < 2^39 looplimit (2^39 - (2049 * 8193^2)) do
            b = b+1;
        end;
        r = (b+1) DIVRM 2;
        // Round to nearest
        assert 4096 <= r && r < 8192;
    end;
    return r;
end;

```


Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// =====

func FPSqrt{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  assert N IN {16,32,64};
  var (fptype,sign,value) : (FType, bit, real) = FPUnpack{N}(op, fpcr);

  var result : bits(N);
  if fptype == FType_SNaN || fptype == FType_QNaN then
    result = FPProcessNaN{N}(fptype, op, fpcr);
  elsif fptype == FType_Zero then
    result = FPZero{N}(sign);
  elsif fptype == FType_Infinity && sign == '0' then
    result = FPInfinity{N}(sign);
  elsif sign == '1' then
    result = FPDefaultNaN{N}(fpcr);
    FPProcessException(FPExc_InvalidOp, fpcr);
  else
    var prec : integer;
    if N == 16 then
      prec = 13; // 10 fraction bits + 3
    elsif N == 32 then
      prec = 26; // 23 fraction bits + 3
    else // N == 64
      prec = 55; // 52 fraction bits + 3
    end;
    value = SqrtRounded(value, prec);
    result = FPRound{N}(value, fpcr);
    FPProcessDenorm(fptype, N, fpcr);
  end;
  return result;
end;
```

Library pseudocode for shared/functions/float/fpsub/FPSub

```
// FPSub()
// =====

func FPSub{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
  let fpxc : boolean = TRUE;          // Generate floating-point exceptions
  return FPSub{N}(op1, op2, fpcr, fpxc);
end;

// FPSub()
// =====

func FPSub{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type, fpxc : boolean) => bits(N)
begin
  assert N IN {16,32,64};
  let rounding : FPRounding = FPRoundingMode(fpcr);

  let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr, fpxc);
  let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr, fpxc);

  var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr, fpxc);
  if !done then
    let inf1 : boolean = (type1 == FType\_Infinity);
    let inf2 : boolean = (type2 == FType\_Infinity);
    let zero1 : boolean = (type1 == FType\_Zero);
    let zero2 : boolean = (type2 == FType\_Zero);

    if inf1 && inf2 && sign1 == sign2 then
      result = FPDefaultNaN{N}(fpcr);
      if fpxc then FPProcessException(FPExc\_InvalidOp, fpcr); end;
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
      result = FPInfinity{N}('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
      result = FPInfinity{N}('1');
    elsif zero1 && zero2 && sign1 == NOT(sign2) then
      result = FPZero{N}(sign1);
    else
      let result_value : real = value1 - value2;
      if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        let result_sign : bit = if rounding == FPRounding\_NEGINF then '1' else '0';
        result = FPZero{N}(result_sign);
      else
        result = FPRound{N}(result_value, fpcr, rounding, fpxc);
      end;
    end;
  end;

  if fpxc then FPProcessDenorms(type1, type2, N, fpcr); end;
end;
return result;
end;
```

Library pseudocode for shared/functions/float/fpsub/FPSub_ZA

```
// FPSub_ZA()
// =====
// Calculates op1-op2 for SME2 ZA-targeting instructions.

func FPSub_ZA{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
  var fpcr : FPCR_Type = fpcr_in;
  let fpxc : boolean = FALSE; // Do not generate floating-point exceptions
  fpcr.DN = '1';              // Generate default NaN values
  return FPSub{N}(op1, op2, fpcr, fpxc);
end;
```

Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =====

func FPThree{N}(sign : bit) => bits(N)
begin
  assert N IN {16,32,64};
  let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
  let F : integer{} = N - (E + 1);
  let exp : bits(E) = '1'::Zeros{E-1};
  let frac : bits(F) = '1'::Zeros{F-1};
  let result : bits((F + E) + 1) = sign :: exp :: frac;

  return result;
end;
```

Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// =====

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

func FPToFixed{M, N}(op : bits(N), fbits : integer, unsigned : boolean, fpcr : FPCR_Type,
    rounding : FPRounding) => bits(M)
begin
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding\_ODD;

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    let altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    let fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    var (fptype,sign,value) : (FType, bit, real) = FPUntpack{N}(op, fpcr, fpexc);

    // If NaN, set cumulative flag or take exception.
    if fptype == FType\_SNaN || fptype == FType\_QNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);
    end;

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    var int_result : integer = RoundDown(value);
    let error : real = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    var round_up : boolean;
    case rounding of
        when FPRounding\_TIEEVEN =>
            round_up = (error > 0.5 || (error == 0.5 && int_result[0] == '1'));
        when FPRounding\_POSINF =>
            round_up = (error != 0.0);
        when FPRounding\_NEGINF =>
            round_up = FALSE;
        when FPRounding\_ZERO =>
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding\_TIEAWAY =>
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
    end;

    if round_up then int_result = int_result + 1; end;

    // Generate saturated result and exceptions.
    let (result, overflow) : (bits(M), boolean) = SatQ{M}(int_result, unsigned);
    if overflow then
        FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc\_Inexact, fpcr);
    end;

    return result;
end;
```

Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =====

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

func FPToFixedJS(op : bits(64), fpcr : FPCR_Type) => (bits(32), bit)
begin

    // If FALSE, never generate Input Denormal floating-point exceptions.
    let fpexc_idenorm : boolean = !(IsFeatureImplemented(FEAT_AFP) &&
                                   !UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    let (fptype, sign, value) = FPUntpack{64}(op, fpcr, fpexc_idenorm);

    var z : bit = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';
    end;

    var int_result = RoundDown(value);
    let error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    let round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1; end;

    var result : integer;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));
    end;

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        z = '0';
    elsif sign == '1' && value == 0.0 then
        z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op[51:0]) then
        z = '0';
    end;

    if fptype == FPType_Infinity then result = 0; end;

    return (result[31:0], z);
end;
```

Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =====

func FPTwo{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = '1'::Zeros{E-1};
    let frac : bits(F) = Zeros{};
    let result : bits((F + E) + 1) = sign :: exp :: frac;
    return result;
end;
```

Library pseudocode for shared/functions/float/fptype/FPType

```
// FPType
// =====

type FPType of enumeration {FPType_Zero,
                             FPType_Denormal,
                             FPType_Nonzero,
                             FPType_Infinity,
                             FPType_QNaN,
                             FPType_SNaN};
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// =====

func FPUnpack{N}(fpval : bits(N), fpcr_in : FPCR_Type) => (FType, bit, real)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    let fpexc : boolean = TRUE;    // Generate floating-point exceptions
    let (fp_type, sign, value) : (FType, bit, real) = FPUnpackBase{N}(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
end;

// FPUnpack()
// =====
//
// Used by data processing, int/fixed to FP and FP to int/fixed conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

func FPUnpack{N}(fpval : bits(N), fpcr_in : FPCR_Type, fpexc : boolean) => (FType, bit, real)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.AHP = '0';
    let (fp_type, sign, value) : (FType, bit, real) = FPUnpackBase{N}(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
end;
```



```

// FPUnpackBase()
// =====

func FPUnpackBase{N}(fpval : bits(N), fpcr : FPCR_Type, fpexc : boolean) => (FType, bit, real)
begin
    let isbfloat16 : boolean = FALSE;
    let (fp_type, sign, value) = FPUnpackBase{N}(fpval, fpcr, fpexc, isbfloat16);
    return (fp_type, sign, value);
end;

// FPUnpackBase()
// =====
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr_in' argument supplies FPCR control bits, 'fpexc' controls the
// generation of floating-point exceptions and 'isbfloat16' determines whether
// N=16 signifies BFloat16 or half-precision type. Status information is updated
// directly in the FPSR where appropriate.

func FPUnpackBase{N}(fpval : bits(N), fpcr_in : FPCR_Type, fpexc : boolean,
    isbfloat16 : boolean) => (FType, bit, real)
begin
    assert N IN {16,32,64};

    let fpcr : FPCR_Type = fpcr_in;

    let altfp : boolean = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32();
    let fiz : boolean = altfp && fpcr.FIZ == '1';
    let fz : boolean = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    var value : real;
    var sign : bit;
    var fptype : FType;

    if N == 16 && !isbfloat16 then
        sign = fpval[15];
        let expl6 : bits(5) = fpval[14:10];
        let frac16 : bits(10) = fpval[9:0];
        if IsZero(expl6) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FType_Zero; value = 0.0;
            else
                fptype = FType_Denormal; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
            end;
        elsif IsOnes(expl6) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FType_Infinity; value = 2.0^1000000;
            else
                fptype = if frac16[9] == '1' then FType_QNaN else FType_SNaN;
                value = 0.0;
            end;
        else
            fptype = FType_Nonzero;
            value = 2.0^(UInt(expl6)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);
        end;

    elsif N == 32 || isbfloat16 then
        var exp32 : bits(8);
        var frac32 : bits(23);
        if isbfloat16 then
            sign = fpval[15];
            exp32 = fpval[14:7];
            frac32 = fpval[6:0] :: Zeros{16};
        else
            sign = fpval[31];
            exp32 = fpval[30:23];
            frac32 = fpval[22:0];
        end;

        if IsZero(exp32) then
            if IsZero(frac32) then
                // Produce zero if value is zero.
                fptype = FType_Zero; value = 0.0;
            elsif fiz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FType_Zero; value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPPProcessException(FPEExc_InputDenorm, fpcr); end;
                end;
            else
                fptype = FType_Denormal; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
            end;
        elsif IsOnes(exp32) then

```

```

    if IsZero(frac32) then
        fptype = FPTYPE\_Infinity; value = 2.0^1000000;
    else
        fptype = if frac32[22] == '1' then FPTYPE\_QNaN else FPTYPE\_SNaN;
        value = 0.0;
    end;
else
    fptype = FPTYPE\_Nonzero;
    value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);
end;

else // N == 64
    sign = fpval[63];
    let exp64 : bits(11) = fpval[62:52];
    let frac64 : bits(52) = fpval[51:0];

    if IsZero(exp64) then
        if IsZero(frac64) then
            // Produce zero if value is zero.
            fptype = FPTYPE\_Zero; value = 0.0;
        elsif fz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
            fptype = FPTYPE\_Zero; value = 0.0;
            // Check whether to raise Input Denormal floating-point exception.
            // fpcr.FIZ==1 does not raise Input Denormal exception.
            if fz then
                // Denormalized input flushed to zero
                if fpexc then FPPProcessException(FPExc\_InputDenorm, fpcr); end;
            end;
        else
            fptype = FPTYPE\_Denormal; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        end;
    elsif IsOnes(exp64) then
        if IsZero(frac64) then
            fptype = FPTYPE\_Infinity; value = 2.0^1000000;
        else
            fptype = if frac64[51] == '1' then FPTYPE\_QNaN else FPTYPE\_SNaN;
            value = 0.0;
        end;
    else
        fptype = FPTYPE\_Nonzero;
        value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
    end;
end;

if sign == '1' then value = -value; end;

return (fptype, sign, value);
end;

```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

```

// FPUnpackCV()
// =====
//
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

func FPUnpackCV{N}(fpval : bits(N), fpcr_in : FPCR_Type) => (FPTYPE, bit, real)
begin
    var fpcr : FPCR_Type = fpcr_in;
    fpcr.FZ16 = '0';
    let fpexc : boolean = TRUE; // Generate floating-point exceptions
    let (fp_type, sign, value) : (FPTYPE, bit, real) = FPUnpackBase{N}(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
end;

```

Library pseudocode for shared/functions/float/fpzzero/FPZero

```

// FPZero()
// =====

func FPZero{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = Zeros{};
    let frac : bits(F) = Zeros{};
    let result : bits((F + E) + 1) = sign :: exp :: frac;
    return result;
end;

```


Library pseudocode for shared/functions/float/vfpexpandimm/VFPEExpandImm

```
// VFPEExpandImm()
// =====

func VFPEExpandImm{N}(imm8 : bits(8)) => bits(N)
begin
  assert N IN {16,32,64};
  let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
  let F : integer{} = (N - E) - 1;
  let sign : bit = imm8[7];
  let exp : bits(E) = NOT(imm8[6])::Replicate{E - 3}(imm8[6])::imm8[5:4];
  let frac : bits(F) = imm8[3:0]::Zeros{F-4};
  let result : bits((F + E) + 1) = sign :: exp :: frac;

  return result;
end;
```

Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags

func AddWithCarry{N}(x : bits(N), y : bits(N), carry_in : bit) => (bits(N), bits(4))
begin
  let unsigned_sum : integer = UInt(x) + UInt(y) + UInt(carry_in);
  let signed_sum : integer = SInt(x) + SInt(y) + UInt(carry_in);
  let result : bits(N) = unsigned_sum[N-1:0]; // same value as signed_sum[N-1:0]
  let n : bit = result[N-1];
  let z : bit = if IsZero(result) then '1' else '0';
  let c : bit = if UInt(result) == unsigned_sum then '0' else '1';
  let v : bit = if SInt(result) == signed_sum then '0' else '1';
  return (result, n::z::c::v);
end;
```

Library pseudocode for shared/functions/interrupts/InterruptID

```
// InterruptID
// =====

type InterruptID of enumeration {
  InterruptID_PMUIRQ,
  InterruptID_COMMIRQ,
  InterruptID_CTIIRQ,
  InterruptID_COMMRX,
  InterruptID_COMMTX,
  InterruptID_CNTP,
  InterruptID_CNTHP,
  InterruptID_CNTHPS,
  InterruptID_CNTPS,
  InterruptID_CNTV,
  InterruptID_CNTHV,
  InterruptID_CNTHVS,
  InterruptID_PMBIRQ,
  InterruptID_HACDBSIRQ,

  InterruptID_TRBIRQ,
};
```

Library pseudocode for shared/functions/interrupts/SetInterruptRequestLevel

```
// SetInterruptRequestLevel()
// =====
// Set a level-sensitive interrupt to the specified level.

impdef func SetInterruptRequestLevel(id : InterruptID, level : Signal)
begin
  return;
end;
```

Library pseudocode for shared/functions/memory/AArch64_BranchAddr

```
// AArch64_BranchAddr()
// =====
// Return the virtual address with tag bits removed.
// This is typically used when the address will be stored to the program counter.

func AArch64_BranchAddr(vaddress : bits(64), el : bits(2)) => bits(64)
begin
    assert !UsingAArch32();
    let msbit : integer{} = AddrTop(vaddress, TRUE, el);
    if msbit == 63 then
        return vaddress;
    elsif (el IN {ELO, EL1} || IsInHost()) && vaddress[msbit] == '1' then
        return SignExtend{64}(vaddress[msbit:0]);
    else
        return ZeroExtend{64}(vaddress[msbit:0]);
    end;
end;
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```
// AccessDescriptor
// =====
// Memory access or translation invocation details that steer architectural behavior

type AccessDescriptor of record {
    acctype : AccessType,
    el : bits(2), // Acting EL for the access
    ss : SecurityState, // Acting Security State for the access
    acqsc : boolean, // Acquire with Sequential Consistency
    acqpc : boolean, // FEAT_LRCPC: Acquire with Processor Consistency
    relsc : boolean, // Release with Sequential Consistency
    limitedordered : boolean, // FEAT_LOR: Acquire/Release with limited ordering
    exclusive : boolean, // Access has Exclusive semantics
    atomicop : boolean, // FEAT_LSE: Atomic read-modify-write access
    modop : MemAtomicOp, // FEAT_LSE: The modification operation in the 'atomicop' access
    nontemporal : boolean, // Hints the access is non-temporal
    read : boolean, // Read from memory or only require read permissions
    write : boolean, // Write to memory or only require write permissions
    cacheop : CacheOp, // DC/IC: Cache operation
    opscope : CacheOpScope, // DC/IC: Scope of cache operation
    cachetype : CacheType, // DC/IC: Type of target cache
    pan : boolean, // FEAT_PAN: The access is subject to PSTATE.PAN
    nonfault : boolean, // SVE: Non-faulting load
    firstfault : boolean, // SVE: First-fault load
    first : boolean, // SVE: First-fault load for the first active element
    contiguous : boolean, // SVE: Contiguous load/store not gather load/scatter store
    predicated : boolean, // SVE: Predicated load/store
    streaming sve : boolean, // SME: Access made by PE while in streaming SVE mode
    ls64 : boolean, // FEAT_LS64: Accesses by accelerator support loads/stores
    withstatus : boolean, // FEAT_LS64: Store with status result
    mops : boolean, // FEAT_MOPS: Memory operation (CPY/SET) accesses
    rcw : boolean, // FEAT_THE: Read-Check-Write access
    rcws : boolean, // FEAT_THE: Read-Check-Write Software access
    toplevel : boolean, // FEAT_THE: Translation table walk access for TTB address
    varange : VARange, // FEAT_THE: The corresponding TTBR supplying the TTB
    a32lsmd : boolean, // A32 Load/Store Multiple Data access
    tagchecked : boolean, // FEAT_MTE2: Access is tag checked
    tagaccess : boolean, // FEAT_MTE: Access targets the tag bits
    stzgm : boolean, // FEAT_MTE: Accesses that store Allocation tags to Device
    // memory are CONSTRAINED UNPREDICTABLE

    Rt : integer, // Register named Rt in the instruction
    Rt2 : integer, // Register named Rt2 in the instruction
    Rs : integer, // Register named Rs in the instruction
    Rs2 : integer, // Register named Rs2 in the instruction
    ispair : boolean, // Access represents a Load/Store pair access
    highestaddressfirst : boolean, // FEAT_LRCPC3: Highest address is accessed first
    lowestaddress : boolean, // Is the current access the lowest address accessed by
    // this instruction
    mpam : MPAMInfo // FEAT_MPAM: MPAM information
};
```

Library pseudocode for shared/functions/memory/AccessType

```
// AccessType
// =====

type AccessType of enumeration {
  AccessType_IFETCH,      // Instruction FETCH
  AccessType_GPR,         // Software load/store to a General Purpose Register
  AccessType_FP,          // Software load/store to an FP register
  AccessType_ASIMD,       // Software ASIMD extension load/store instructions
  AccessType_SVE,         // Software SVE load/store instructions
  AccessType_SME,         // Software SME load/store instructions
  AccessType_IC,          // Sysop IC
  AccessType_DC,          // Sysop DC (not DC {Z,G,GZ}VA)
  AccessType_DCZero,      // Sysop DC {Z,G,GZ}VA
  AccessType_AT,          // Sysop AT
  AccessType_NV2,         // NV2 memory redirected access
  AccessType_SPE,         // Statistical Profiling buffer access
  AccessType_GCS,         // Guarded Control Stack access
  AccessType_TRBE,        // Trace Buffer access
  AccessType_GPTW,        // Granule Protection Table Walk
  AccessType_HACDBS,      // Access to the HACDBS structure
  AccessType_HDBSS,       // Access to entries in HDBSS
  AccessType_TTW          // Translation Table Walk
};
```

Library pseudocode for shared/functions/memory/AddrTop

```
// AddrTop()
// =====
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

func AddrTop(address : bits(64), IsInstr : boolean, el : bits(2)) => AddressSize
begin
  assert HaveEL(el);
  let regime : bits(2) = S1TranslationRegime(el);
  if ELUsingAArch32(regime) then
    // AArch32 translation regime.
    return 31;
  else
    if EffectiveTBI(address, IsInstr, el) == '1' then
      return 55;
    else
      return 63;
    end;
  end;
end;
```

Library pseudocode for shared/functions/memory/AddressSize

```
// AddressSize
// =====

type AddressSize of integer{3..64};
```

Library pseudocode for shared/functions/memory/AlignmentEnforced

```
// AlignmentEnforced()
// =====
// For the active translation regime, determine if alignment is required by all accesses

func AlignmentEnforced() => boolean
begin
  let regime : Regime = TranslationRegime(PSTATE.EL);
  var A : bit;
  case regime of
    when Regime\_EL30 => A = SCTLR() .A;
    when Regime\_EL3 => A = SCTLR\_EL3() .A;
    when Regime\_EL2 => A = if ELUsingAArch32(EL2) then HSCTLR() .A else SCTLR\_EL2() .A;
    when Regime\_EL20 => A = SCTLR\_EL2() .A;
    when Regime\_EL10 => A = if ELUsingAArch32(EL1) then SCTLR() .A else SCTLR\_EL1() .A;
    otherwise => unreachable;
  end;
  return A == '1';
end;
```

Library pseudocode for shared/functions/memory/AllInAlignedQuantity

```
// AllInAlignedQuantity()
// =====
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

readonly func AllInAlignedQuantity{N : integer{32, 64}}(address : bits(N),
                                                         size : integer,
                                                         alignment : integer) => boolean

begin
    return (AlignDownSize(address+(size-1), alignment as AddressSize) ==
            AlignDownSize(address, alignment as AddressSize));
end;
```

Library pseudocode for shared/functions/memory/Allocation

```
// Allocation hints
// =====

constant MemHint_No : bits(2) = '00';    // No Read-Allocate, No Write-Allocate
constant MemHint_WA : bits(2) = '01';    // No Read-Allocate, Write-Allocate
constant MemHint_RA : bits(2) = '10';    // Read-Allocate, No Write-Allocate
constant MemHint_RWA : bits(2) = '11';   // Read-Allocate, Write-Allocate
```

Library pseudocode for shared/functions/memory/BigEndian

```
// BigEndian()
// =====

func BigEndian(acctype : AccessType) => boolean
begin
    var bigend : boolean;
    if IsFeatureImplemented(FEAT_NV2) && acctype == AccessType\_NV2 then
        return SCTLRL_EL2().EE == '1';
    end;

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR_ELx().E0E != '0');
    else
        bigend = (SCTLR_ELx().EE != '0');
    end;
    return bigend;
end;
```

Library pseudocode for shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// =====

func BigEndianReverse{width}(value : bits(width)) => bits(width)
begin
    assert width IN {8, 16, 32, 64, 128, 256};

    if width == 8 then return value; end;
    return Reverse{width}(value, 8);
end;
```

Library pseudocode for shared/functions/memory/CacheOp

```
// CacheOp
// =====

type CacheOp of enumeration {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};
```

Library pseudocode for shared/functions/memory/CacheOpScope

```
// CacheOpScope
// =====

type CacheOpScope of enumeration {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoE,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_PoPA,
    CacheOpScope_PoPS,
    CacheOpScope_OuterCache,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};
```

Library pseudocode for shared/functions/memory/CachePASpace

```
// CachePASpace
// =====

type CachePASpace of enumeration {
    CPAS_NonSecure,
    CPAS_Any,                      // Applicable only for DC *SW / IC IALLU* in Root state:
                                   // match entries from any PA Space
    CPAS_RealmNonSecure,          // Applicable only for DC *SW / IC IALLU* in Realm state:
                                   // match entries from Realm or Non-Secure PAS
    CPAS_Realm,
    CPAS_Root,
    CPAS_SystemAgent,             // Applicable only for DC by PA:
                                   // match entries from the System Agent PAS
    CPAS_NonSecureProtected,      // Applicable only for DC by PA:
                                   // match entries from the Non-Secure Protected PAS
    CPAS_NA6,                     // Reserved
    CPAS_NA7,                     // Reserved
    CPAS_SecureNonSecure,         // Applicable only for DC *SW / IC IALLU* in Secure state:
                                   // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
```

Library pseudocode for shared/functions/memory/CacheType

```
// CacheType
// =====

type CacheType of enumeration {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};
```

Library pseudocode for shared/functions/memory/Cacheability

```
// Cacheability attributes
// =====

constant MemAttr_NC : bits(2) = '00';    // Non-cacheable
constant MemAttr_WT : bits(2) = '10';    // Write-through
constant MemAttr_WB : bits(2) = '11';    // Write-back
```

Library pseudocode for shared/functions/memory/CreateAccDescA32LSMD

```
// CreateAccDescA32LSMD()
// =====
// Access descriptor for A32 loads/store multiple general purpose registers

func CreateAccDescA32LSMD(memop : MemOp) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_GPR);

    accdesc.read      = memop == MemOp\_LOAD;
    accdesc.write     = memop == MemOp\_STORE;
    accdesc.pan       = TRUE;
    accdesc.a32lsmd   = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescASIMD

```
// CreateAccDescASIMD()
// =====
// Access descriptor for ASIMD&FP loads/stores

func CreateAccDescASIMD(memop : MemOp, nontemporal : boolean, tagchecked : boolean,
    privileged : boolean) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    return CreateAccDescASIMD(memop, nontemporal, tagchecked, privileged, ispair);
end;

// CreateAccDescASIMD()
// =====

func CreateAccDescASIMD(memop : MemOp, nontemporal : boolean, tagchecked : boolean,
    privileged : boolean, ispair : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_ASIMD);

    accdesc.nontemporal      = nontemporal;
    accdesc.el              = if !privileged then ELO else PSTATE.EL;
    accdesc.read             = memop == MemOp\_LOAD;
    accdesc.write            = memop == MemOp\_STORE;
    accdesc.ispair           = ispair;
    accdesc.pan              = TRUE;
    accdesc.streamingsve     = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    end;
    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescASIMDAcqRel

```
// CreateAccDescASIMDAcqRel()
// =====
// Access descriptor for ASIMD&FP loads/stores with ordering semantics

func CreateAccDescASIMDAcqRel(memop : MemOp, tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_ASIMD);

    accdesc.acqpc            = memop == MemOp\_LOAD;
    accdesc.relsc            = memop == MemOp\_STORE;
    accdesc.read             = memop == MemOp\_LOAD;
    accdesc.write            = memop == MemOp\_STORE;
    accdesc.pan              = TRUE;
    accdesc.streamingsve     = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    end;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescAT

```
// CreateAccDescAT()
// =====
// Access descriptor for address translation operations

func CreateAccDescAT(ss : SecurityState, el : bits(2), ataccess : ATAccess) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_AT);

    accdesc.el          = el;
    accdesc.ss          = ss;
    if ImpDefBool("MPAM uses the EL targeted by the AT instruction") then
        accdesc.mpam = GenMPAMatEL(AccessType\_AT, el);
    end;
    case ataccess of
        when ATAccess\_Read =>
            accdesc.(read, write, pan) = (TRUE, FALSE, FALSE);
        when ATAccess\_ReadPAN =>
            accdesc.(read, write, pan) = (TRUE, FALSE, TRUE);
        when ATAccess\_Write =>
            accdesc.(read, write, pan) = (FALSE, TRUE, FALSE);
        when ATAccess\_WritePAN =>
            accdesc.(read, write, pan) = (FALSE, TRUE, TRUE);
        when ATAccess\_Any =>
            accdesc.(read, write, pan) = (FALSE, FALSE, FALSE);
    end;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescAcqRel

```
// CreateAccDescAcqRel()
// =====
// Access descriptor for general purpose register loads/stores with ordering semantics

func CreateAccDescAcqRel(memop : MemOp, tagchecked : boolean, acqsc : boolean) => AccessDescriptor
begin
    let Rt : integer = -1;
    return CreateAccDescAcqRel(memop, tagchecked, acqsc, Rt);
end;

func CreateAccDescAcqRel(memop : MemOp, tagchecked : boolean,
                        acqsc : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescAcqRel(memop, tagchecked, ispair, acqsc, Rt, Rt2);
end;

func CreateAccDescAcqRel(memop : MemOp, tagchecked : boolean, ispair : boolean,
                        acqsc : boolean, Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_GPR);

    accdesc.acqsc          = acqsc;
    accdesc.rels           = memop == MemOp\_STORE;
    accdesc.read            = memop == MemOp\_LOAD;
    accdesc.write           = memop == MemOp\_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.ispair          = ispair;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescAtomicOp

```
// CreateAccDescAtomicOp()
// =====
// Access descriptor for atomic read-modify-write memory accesses

func CreateAccDescAtomicOp(modop : MemAtomicOp, acquire : boolean, release : boolean,
    tagchecked : boolean, privileged : boolean, Rt : integer,
    Rs : integer) => AccessDescriptor

begin
    let Rt2 : integer = -1;
    let Rs2 : integer = -1;

    return CreateAccDescAtomicOp(modop, acquire, release, tagchecked, privileged, Rt, Rt2, Rs, Rs2);
end;

func CreateAccDescAtomicOp(modop : MemAtomicOp, acquire : boolean, release : boolean,
    tagchecked : boolean, privileged : boolean, Rt : integer,
    Rt2 : integer, Rs : integer, Rs2 : integer) => AccessDescriptor

begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_GPR);

    accdesc.acqsc      = acquire;
    accdesc.el        = if !privileged then ELO else PSTATE.EL;
    accdesc.relsc     = release;
    accdesc.atomicop  = TRUE;
    accdesc.modop     = modop;
    accdesc.read      = TRUE;
    accdesc.write     = TRUE;
    accdesc.pan       = TRUE;
    accdesc.tagchecked = tagchecked;
    accdesc.Rs        = Rs;
    accdesc.Rs2       = Rs2;
    accdesc.Rt        = Rt;
    accdesc.Rt2       = Rt2;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescDC

```
// CreateAccDescDC()
// =====
// Access descriptor for data cache operations

func CreateAccDescDC(cache : CacheRecord) => AccessDescriptor

begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_DC);

    accdesc.cacheop    = cache.cacheop;
    accdesc.cachetype  = cache.cachetype;
    accdesc.opscope    = cache.opscope;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescDCZero

```
// CreateAccDescDCZero()
// =====
// Access descriptor for data cache zero operations

func CreateAccDescDCZero(cachetype : CacheType) => AccessDescriptor

begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_DCZero);

    accdesc.write      = TRUE;
    accdesc.pan        = TRUE;
    accdesc.tagchecked  = cachetype == CacheType\_Data;
    accdesc.tagaccess   = cachetype IN {CacheType\_Tag,
    CacheType\_Data\_Tag};
    accdesc.cachetype  = cachetype;

    return accdesc;
end;
```


Library pseudocode for shared/functions/memory/CreateAccDescExLDST

```
// CreateAccDescExLDST()
// =====
// Access descriptor for general purpose register loads/stores with exclusive semantics

func CreateAccDescExLDST(memop : MemOp, acqrel : boolean, tagchecked : boolean,
                        privileged : boolean) => AccessDescriptor
begin
    let Rt : integer = -1;
    return CreateAccDescExLDST(memop, acqrel, tagchecked, privileged, Rt);
end;

// CreateAccDescExLDST()
// =====

func CreateAccDescExLDST(memop : MemOp, acqrel : boolean, tagchecked : boolean,
                        privileged : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescExLDST(memop, acqrel, tagchecked, privileged, ispair, Rt, Rt2);
end;

// CreateAccDescExLDST()
// =====

func CreateAccDescExLDST(memop : MemOp, acqrel : boolean, tagchecked : boolean,
                        privileged : boolean, ispair : boolean,
                        Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_GPR);

    accdesc.acqsc      = acqrel && memop == MemOp\_LOAD;
    accdesc.relsc      = acqrel && memop == MemOp\_STORE;
    accdesc.exclusive  = TRUE;
    accdesc.el         = if !privileged then EL0 else PSTATE.EL;
    accdesc.read       = memop == MemOp\_LOAD;
    accdesc.write      = memop == MemOp\_STORE;
    accdesc.ispair     = ispair;
    accdesc.pan        = TRUE;
    accdesc.tagchecked = tagchecked;
    accdesc.Rt         = Rt;
    accdesc.Rt2        = Rt2;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescFPAtomicOp

```
// CreateAccDescFPAtomicOp()
// =====
// Access descriptor for FP atomic read-modify-write memory accesses

func CreateAccDescFPAtomicOp(modop : MemAtomicOp, acquire : boolean, release : boolean,
                            tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_FP);

    accdesc.acqsc      = acquire;
    accdesc.relsc      = release;
    accdesc.atomicop    = TRUE;
    accdesc.modop       = modop;
    accdesc.read        = TRUE;
    accdesc.write       = TRUE;
    accdesc.pan         = TRUE;
    accdesc.streamingsve = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    end;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescGCS

```
// CreateAccDescGCS()
// =====
// Access descriptor for memory accesses to the Guarded Control Stack

func CreateAccDescGCS(memop : MemOp, privileged : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GCS);

    accdesc.el          = if !privileged then EL0 else PSTATE.EL;
    accdesc.read        = memop == MemOp_LOAD;
    accdesc.write       = memop == MemOp_STORE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescGCSSS1

```
// CreateAccDescGCSSS1()
// =====
// Access descriptor for memory accesses to the Guarded Control Stack that switch stacks

func CreateAccDescGCSSS1(privileged : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GCS);

    accdesc.el          = if !privileged then EL0 else PSTATE.EL;
    accdesc.atomicop    = TRUE;
    accdesc.modop       = MemAtomicOp_GCSSS1;
    accdesc.read        = TRUE;
    accdesc.write       = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescGPR

```
// CreateAccDescGPR()
// =====
// Access descriptor for general purpose register loads/stores
// without exclusive or ordering semantics

func CreateAccDescGPR(memop : MemOp, nontemporal : boolean, privileged : boolean,
    tagchecked : boolean) => AccessDescriptor
begin
    let Rt : integer = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, tagchecked, Rt);
end;

func CreateAccDescGPR(memop : MemOp, nontemporal : boolean, privileged : boolean,
    tagchecked : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, tagchecked, ispair, Rt, Rt2);
end;

func CreateAccDescGPR(memop : MemOp, nontemporal : boolean, privileged : boolean,
    tagchecked : boolean, ispair : boolean,
    Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.el          = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal = nontemporal;
    accdesc.read        = memop == MemOp_LOAD;
    accdesc.write       = memop == MemOp_STORE;
    accdesc.pan         = TRUE;
    accdesc.tagchecked  = tagchecked;
    accdesc.Rt          = Rt;
    accdesc.Rt2         = Rt2;
    accdesc.ispair      = ispair;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescGPTW

```
// CreateAccDescGPTW()
// =====
// Access descriptor for Granule Protection Table walks

func CreateAccDescGPTW(accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_GPTW, accdesc_in.mpam);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.read        = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescHACDBS

```
// CreateAccDescHACDBS()
// =====
// Access descriptor for memory accesses to the HACDBS structure.

func CreateAccDescHACDBS() => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_HACDBS);

    accdesc.read        = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescHDBSS

```
// CreateAccDescHDBSS()
// =====
// Access descriptor for appending entries to the HDBSS

func CreateAccDescHDBSS(accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_HDBSS, accdesc_in.mpam);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.write       = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescIC

```
// CreateAccDescIC()
// =====
// Access descriptor for instruction cache operations

func CreateAccDescIC(cache : CacheRecord) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_IC);

    accdesc.cacheop     = cache.cacheop;
    accdesc.cachetype    = cache.cachetype;
    accdesc.opscope     = cache.opscope;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescIFetch

```
// CreateAccDescIFetch()
// =====
// Access descriptor for instruction fetches

func CreateAccDescIFetch() => AccessDescriptor
begin
    let accdesc : AccessDescriptor = NewAccDesc(AccessType\_IFETCH);

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescLDAcqPC

```
// CreateAccDescLDAcqPC()
// =====
// Access descriptor for general purpose register loads with local ordering semantics

func CreateAccDescLDAcqPC(tagchecked : boolean, acquire : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescLDAcqPC(tagchecked, ispair, acquire, Rt, Rt2);
end;

func CreateAccDescLDAcqPC(tagchecked : boolean, ispair : boolean, acquire : boolean,
    Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_GPR);

    accdesc.acqpc      = acquire;
    accdesc.read       = TRUE;
    accdesc.pan        = TRUE;
    accdesc.tagchecked = tagchecked;
    accdesc.ispair     = ispair;
    accdesc.Rt         = Rt;
    accdesc.Rt2        = Rt2;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescLDGSTG

```
// CreateAccDescLDGSTG()
// =====
// Access descriptor for tag memory loads/stores

func CreateAccDescLDGSTG(memop : MemOp, stzgm : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescLDGSTG(memop, stzgm, ispair, Rt, Rt2);
end;

// CreateAccDescLDGSTG()
// =====

func CreateAccDescLDGSTG(memop : MemOp, stzgm : boolean, ispair : boolean, Rt : integer,
    Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_GPR);

    accdesc.read       = memop == MemOp\_LOAD;
    accdesc.write      = memop == MemOp\_STORE;
    accdesc.pan        = TRUE;
    accdesc.tagaccess  = TRUE;
    accdesc.stzgm      = stzgm;
    accdesc.ispair     = ispair;
    accdesc.Rt         = Rt;
    accdesc.Rt2        = Rt2;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescLOR

```
// CreateAccDescLOR()
// =====
// Access descriptor for general purpose register loads/stores with limited ordering semantics

func CreateAccDescLOR(memop : MemOp, tagchecked : boolean,
    acqsc : boolean, Rt : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_GPR);

    accdesc.acqsc      = acqsc;
    accdesc.relsc      = memop == MemOp\_STORE;
    accdesc.limitedordered = TRUE;
    accdesc.read       = memop == MemOp\_LOAD;
    accdesc.write      = memop == MemOp\_STORE;
    accdesc.pan        = TRUE;
    accdesc.tagchecked = tagchecked;
    accdesc.Rt         = Rt;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescLS64

```
// CreateAccDescLS64 ()
// =====
// Access descriptor for accelerator-supporting memory accesses

func CreateAccDescLS64(memop : MemOp, withstatus : boolean,
    tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.pan           = TRUE;
    accdesc.ls64          = TRUE;
    accdesc.withstatus    = withstatus;
    accdesc.tagchecked    = tagchecked;
    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescMOPS

```
// CreateAccDescMOPS ()
// =====
// Access descriptor for data memory copy and set instructions

func CreateAccDescMOPS(memop : MemOp, privileged : boolean,
    nontemporal : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.el            = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal    = nontemporal;
    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.pan           = TRUE;
    accdesc.mops          = TRUE;
    accdesc.tagchecked    = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescNV2

```
// CreateAccDescNV2 ()
// =====
// Access descriptor nested virtualization memory indirection loads/stores

func CreateAccDescNV2(memop : MemOp) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_NV2);

    accdesc.el            = EL2;
    accdesc.ss            = SecurityStateAtEL(EL2);
    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescRCW

```
// CreateAccDescRCW()
// =====
// Access descriptor for atomic read-check-write memory accesses

func CreateAccDescRCW(modop : MemAtomicOp, soft : boolean, acquire : boolean,
    release : boolean, tagchecked : boolean,
    Rt : integer, Rs : integer) => AccessDescriptor

begin
    let Rt2 : integer = -1;
    let Rs2 : integer = -1;

    return CreateAccDescRCW(modop, soft, acquire, release, tagchecked, Rt, Rt2, Rs, Rs2);
end;

func CreateAccDescRCW(modop : MemAtomicOp, soft : boolean, acquire : boolean,
    release : boolean, tagchecked : boolean, Rt : integer, Rt2 : integer,
    Rs : integer, Rs2 : integer) => AccessDescriptor

begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_GPR);

    accdesc.acqsc      = acquire;
    accdesc.relsc      = release;
    accdesc.rcw        = TRUE;
    accdesc.rcws       = soft;
    accdesc.atomicop   = TRUE;
    accdesc.modop      = modop;
    accdesc.read       = TRUE;
    accdesc.write      = TRUE;
    accdesc.pan        = TRUE;
    accdesc.tagchecked = tagchecked;
    accdesc.Rt         = Rt;
    accdesc.Rt2        = Rt2;
    accdesc.Rs         = Rs;
    accdesc.Rs2        = Rs2;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescS1TTW

```
// CreateAccDescS1TTW()
// =====
// Access descriptor for stage 1 translation table walks

func CreateAccDescS1TTW(toplevel : boolean,
    varange : VARange,
    accdesc_in : AccessDescriptor) => AccessDescriptor

begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_TTW, accdesc_in.mpam);

    accdesc.el        = accdesc_in.el;
    accdesc.ss        = accdesc_in.ss;
    accdesc.read      = TRUE;
    accdesc.toplevel  = toplevel;
    accdesc.varange   = varange;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescS2TTW

```
// CreateAccDescS2TTW()
// =====
// Access descriptor for stage 2 translation table walks

func CreateAccDescS2TTW(accdesc_in : AccessDescriptor) => AccessDescriptor

begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_TTW, accdesc_in.mpam);

    accdesc.el        = accdesc_in.el;
    accdesc.ss        = accdesc_in.ss;
    accdesc.read      = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescSME

```
// CreateAccDescSME()
// =====
// Access descriptor for SME loads/stores

func CreateAccDescSME(memop : MemOp, nontemporal : boolean, contiguous : boolean,
    tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_SME);

    accdesc.nontemporal      = nontemporal;
    accdesc.read             = memop == MemOp_LOAD;
    accdesc.write            = memop == MemOp_STORE;
    accdesc.pan              = TRUE;
    accdesc.contiguous       = contiguous;
    accdesc.streamingsve     = TRUE;
    if ImpDefBool("No tag checking of SME LDR & STR instructions") then
        accdesc.tagchecked   = FALSE;
    else
        accdesc.tagchecked   = tagchecked;
    end;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescSPE

```
// CreateAccDescSPE()
// =====
// Access descriptor for memory accesses by Statistical Profiling unit

func CreateAccDescSPE(owning_ss : SecurityState, owning_el : bits(2)) => AccessDescriptor
begin
    let mpaminfo : MPAMInfo = GenMPAMAtEL(AccessType_SPE, owning_el);
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_SPE, mpaminfo);

    accdesc.el              = owning_el;
    accdesc.ss              = owning_ss;
    accdesc.write            = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescSTGMOPS

```
// CreateAccDescSTGMOPS()
// =====
// Access descriptor for tag memory set instructions

func CreateAccDescSTGMOPS(privileged : boolean, nontemporal : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then ELO else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.mops            = TRUE;
    accdesc.tagaccess       = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescSVE

```
// CreateAccDescSVE()
// =====
// Access descriptor for general SVE loads/stores

func CreateAccDescSVE(memop : MemOp, nontemporal : boolean, contiguous : boolean,
    tagchecked : boolean) => AccessDescriptor
begin
    let predicated : boolean = FALSE;
    return CreateAccDescSVE(memop, nontemporal, contiguous, predicated, tagchecked);
end;

func CreateAccDescSVE(memop : MemOp, nontemporal : boolean, contiguous : boolean,
    predicated : boolean, tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_SVE);

    accdesc.nontemporal      = nontemporal;
    accdesc.read             = memop == MemOp\_LOAD;
    accdesc.write            = memop == MemOp\_STORE;
    accdesc.pan              = TRUE;
    accdesc.contiguous       = contiguous;
    accdesc.streamingsve     = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked   = FALSE;
    else
        accdesc.tagchecked   = tagchecked;
    end;
    accdesc.predicated       = predicated;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescSVEFF

```
// CreateAccDescSVEFF()
// =====
// Access descriptor for first-fault SVE loads

func CreateAccDescSVEFF(contiguous : boolean, tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_SVE);

    accdesc.read             = TRUE;
    accdesc.pan              = TRUE;
    accdesc.firstfault       = TRUE;
    accdesc.first            = TRUE;
    accdesc.contiguous       = contiguous;
    accdesc.streamingsve     = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked   = FALSE;
    else
        accdesc.tagchecked   = tagchecked;
    end;
    accdesc.predicated       = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescSVENF

```
// CreateAccDescSVENF()
// =====
// Access descriptor for non-fault SVE loads

func CreateAccDescSVENF(contiguous : boolean, tagchecked : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_SVE);

    accdesc.read             = TRUE;
    accdesc.pan              = TRUE;
    accdesc.nonfault         = TRUE;
    accdesc.contiguous       = contiguous;
    accdesc.streamingsve     = InStreamingMode();
    if (accdesc.streamingsve && ImpDefBool(
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode")) then
        accdesc.tagchecked   = FALSE;
    else
        accdesc.tagchecked   = tagchecked;
    end;
    accdesc.predicated       = TRUE;

    return accdesc;
end;
```


Library pseudocode for shared/functions/memory/CreateAccDescTRBE

```
// CreateAccDescTRBE()
// =====
// Access descriptor for memory accesses by Trace Buffer Unit

func CreateAccDescTRBE(owning_ss : SecurityState, owning_el : bits(2)) => AccessDescriptor
begin
    var mpam : MPAMInfo;
    if SelfHostedTraceEnabled() then
        mpam = GenMPAMatEL(AccessType\_TRBE, owning_el);
    else
        mpam = GenMPAMCurEL(AccessType\_TRBE);
    end;
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_TRBE, mpam);

    accdesc.el          = owning_el;
    accdesc.ss          = owning_ss;
    accdesc.write       = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/CreateAccDescTTEUpdate

```
// CreateAccDescTTEUpdate()
// =====
// Access descriptor for translation table entry HW update

func CreateAccDescTTEUpdate(accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType\_TTW, accdesc_in.mpam);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.atomicop    = TRUE;
    accdesc.modop       = MemAtomicOp\_CAS;
    accdesc.read        = TRUE;
    accdesc.write       = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
// DataMemoryBarrier()
// =====

impdef func DataMemoryBarrier(types : MBReqTypes)
begin
    return;
end;
```

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
// DataSynchronizationBarrier()
// =====

impdef func DataSynchronizationBarrier(scope : MBMaintenanceScope,
                                       types : MBReqTypes,
                                       nXS : boolean)
begin
    return;
end;
```

Library pseudocode for shared/functions/memory/DeviceType

```
// DeviceType
// =====
// Extended memory types for Device memory.

type DeviceType of enumeration {DeviceType_GRE, DeviceType_nGRE,
                                DeviceType_nGnRE, DeviceType_nGnRnE};
```

Library pseudocode for shared/functions/memory/EffectiveMTX

```
// EffectiveMTX()
// =====
// Returns the effective MTX in the AArch64 stage 1 translation regime for "el".

func EffectiveMTX(address : bits(64), is_instr : boolean, el : bits(2)) => bit
begin
    var mtx : bit;
    assert HaveEL(el);
    let regime : bits(2) = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    if is_instr then
        mtx = '0';
    else
        case regime of
            when EL1 =>
                mtx = if address[55] == '1' then TCR_EL1().MTX1 else TCR_EL1().MTX0;
            when EL2 =>
                if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                    mtx = if address[55] == '1' then TCR_EL2().MTX1 else TCR_EL2().MTX0;
                else
                    mtx = TCR_EL2().MTX;
                end;
            when EL3 =>
                mtx = TCR_EL3().MTX;
        end;
    end;

    return mtx;
end;
```

Library pseudocode for shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// =====
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

func EffectiveTBI(address : bits(64), IsInstr : boolean, el : bits(2)) => bit
begin
    var tbi : bit;
    var tbid : bit;
    assert HaveEL(el);
    let regime : bits(2) = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1 =>
            tbi = if address[55] == '1' then TCR_EL1().TBI1 else TCR_EL1().TBI0;
            if IsFeatureImplemented(FEAT_PAuth) then
                tbid = if address[55] == '1' then TCR_EL1().TBID1 else TCR_EL1().TBID0;
            end;
        when EL2 =>
            if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                tbi = if address[55] == '1' then TCR_EL2().TBI1 else TCR_EL2().TBI0;
                if IsFeatureImplemented(FEAT_PAuth) then
                    tbid = if address[55] == '1' then TCR_EL2().TBID1 else TCR_EL2().TBID0;
                end;
            else
                tbi = TCR_EL2().TBI;
                if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL2().TBID; end;
            end;
        when EL3 =>
            tbi = TCR_EL3().TBI;
            if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL3().TBID; end;
    end;

    return (if (tbi == '1' && (!IsFeatureImplemented(FEAT_PAuth) || tbid == '0' || !IsInstr)) then '1' else '0');
end;
```

Library pseudocode for shared/functions/memory/EffectiveTCMA

```
// EffectiveTCMA()
// =====
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

func EffectiveTCMA(address : bits(64), el : bits(2)) => bit
begin
    var tcma : bit;
    assert HaveEL(el);
    let regime : bits(2) = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1 =>
            tcma = if address[55] == '1' then TCR_EL1().TCMA1 else TCR_EL1().TCMA0;
        when EL2 =>
            if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                tcma = if address[55] == '1' then TCR_EL2().TCMA1 else TCR_EL2().TCMA0;
            else
                tcma = TCR_EL2().TCMA;
            end;
        when EL3 =>
            tcma = TCR_EL3().TCMA;
    end;

    return tcma;
end;
```

Library pseudocode for shared/functions/memory/ErrorState

```
// ErrorState
// =====
// The allowed error states that can be returned by memory and used by the PE.

type ErrorState of enumeration {
    ErrorState_UC,           // Uncontainable
    ErrorState_UEU,          // Unrecoverable state
    ErrorState_UEO,          // Restartable state
    ErrorState_UER,          // Recoverable state
    ErrorState_CE};          // Corrected
```

Library pseudocode for shared/functions/memory/Fault

```
// Fault
// =====
// Fault types.

type Fault of enumeration {
    Fault_None,
    Fault_AccessFlag,
    Fault_Alignment,
    Fault_Background,
    Fault_Domain,
    Fault_Permission,
    Fault_Translation,
    Fault_AddressSize,
    Fault_SyncExternal,
    Fault_SyncExternalOnWalk,
    Fault_SyncParity,
    Fault_SyncParityOnWalk,
    Fault_GPCFOnWalk,
    Fault_GPCFOnOutput,
    Fault_AsyncParity,
    Fault_AsyncExternal,
    Fault_TagCheck,
    Fault_Debug,
    Fault_TLBConflict,
    Fault_BranchTarget,
    Fault_UnsupportedAtomicHWUpdate,
    Fault_Lockdown,
    Fault_Exclusive,
    Fault_ICacheMaint};
```

Library pseudocode for shared/functions/memory/FaultRecord

```
// FaultRecord
// =====
// Fields that relate only to Faults.

type FaultRecord of record {
    statuscode : Fault,           // Fault Status
    accessdesc : AccessDescriptor, // Details of the faulting access
    vaddress   : bits(64),         // Faulting virtual address
    ipaddress  : FullAddress,       // Intermediate physical address
    paddress   : FullAddress,       // Physical address
    gpcf       : GPCFRecord,        // Granule Protection Check Fault record
    gpcfs2walk : boolean,          // GPC for a stage 2 translation table walk
    s2fslwalk  : boolean,          // Is on a Stage 1 translation table walk
    write      : boolean,          // TRUE for a write, FALSE for a read
    sltagnotdata : boolean,        // TRUE for a fault due to tag not accessible at stage 1.
    tagaccess  : boolean,          // TRUE for a fault due to NoTagAccess permission.
    level      : integer,          // For translation, access flag and Permission faults
    extflag    : bit,              // IMPLEMENTATION DEFINED syndrome for External aborts
    secondstage : boolean,         // Is a Stage 2 abort
    assuredonly : boolean,         // Stage 2 Permission fault due to AssuredOnly attribute
    toplevel    : boolean,         // Stage 2 Permission fault due to TopLevel
    overlay     : boolean,         // Fault due to overlay permissions
    dirtybit    : boolean,         // Fault due to dirty state
    domain      : bits(4),         // Domain number, AArch32 only
    merrorstate : ErrorState,        // Incoming error state from memory
    hdbssf      : boolean,         // Fault caused by HDBSS
    watchptinfo : WatchpointInfo,   // Watchpoint related fields
    debugmoe    : bits(4)         // Debug method of entry, from AArch32 only
};
```

Library pseudocode for shared/functions/memory/FullAddress

```
// FullAddress
// =====
// Physical or Intermediate Physical Address type.
// Although AArch32 only has access to 40 bits of physical or intermediate physical address space,
// the full address type has NUM_PABITS bits to allow interprocessing with AArch64.
// The maximum physical or intermediate physical address size is IMPLEMENTATION DEFINED,
// but never exceeds NUM_PABITS bits.

type FullAddress of record {
    paspace : PASpace,
    address  : bits(NUM\_PABITS)
};
```

Library pseudocode for shared/functions/memory/GPCF

```
// GPCF
// ====
// Possible Granule Protection Check Fault reasons

type GPCF of enumeration {
    GPCF_None,           // No fault
    GPCF_AddressSize,    // GPT address size fault
    GPCF_Walk,           // GPT walk fault
    GPCF_EABT,           // Synchronous External abort on GPT fetch
    GPCF_Fail            // Granule protection fault
};
```

Library pseudocode for shared/functions/memory/GPCFRecord

```
// GPCFRecord
// =====
// Full details of a Granule Protection Check Fault

type GPCFRecord of record {
    gpf : GPCF,
    level : integer
};
```

Library pseudocode for shared/functions/memory/Hint_Prefetch

```
// Hint_Prefetch()
// =====
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.

impdef func Hint_Prefetch(address : bits(64),
                        hint : PrefetchHint,
                        target : integer,
                        stream : boolean)
begin
    return;
end;
```

Library pseudocode for shared/functions/memory/Hint_RangePrefetch

```
// Hint_RangePrefetch()
// =====
// Signals the memory system that data memory accesses from a specified range
// of addresses are likely to occur in the near future. The memory system can
// respond by taking actions that are expected to speed up the memory accesses
// when they do occur, such as preloading the locations within the specified
// address ranges into one or more caches.

impdef func Hint_RangePrefetch(address : bits(64), length : integer, stride : integer,
                              count : integer, reuse : integer, operation : bits(6))
begin
    return;
end;
```

Library pseudocode for shared/functions/memory/IsContiguousSVEAccess

```
// IsContiguousSVEAccess()
// =====
// Return TRUE if memory access is contiguous load/stores in an SVE mode.

func IsContiguousSVEAccess(accdesc : AccessDescriptor) => boolean
begin
    return accdesc.acctype == AccessType\_SVE && accdesc.contiguous;
end;
```

Library pseudocode for shared/functions/memory/IsRelaxedWatchpointAccess

```
// IsRelaxedWatchpointAccess()
// =====
// Return TRUE if memory access is one of -
// - SIMD&FP load/store instruction when the PE is in Streaming SVE mode
// - SVE contiguous vector load/store instruction.
// - SME load/store instruction

func IsRelaxedWatchpointAccess(accdesc : AccessDescriptor) => boolean
begin
    return (IsContiguousSVEAccess(accdesc) ||
           IsSMEAccess(accdesc) ||
           (IsSIMDFPAccess(accdesc) && InStreamingMode()));
end;
```

Library pseudocode for shared/functions/memory/IsSIMDFPAccess

```
// IsSIMDFPAccess()
// =====
// Return TRUE if access is SIMD&FP.

func IsSIMDFPAccess(accdesc : AccessDescriptor) => boolean
begin
    return accdesc.acctype == AccessType\_ASIMD;
end;
```

Library pseudocode for shared/functions/memory/IsSMEAccess

```
// IsSMEAccess()
// =====
// Return TRUE if access is of SME load/stores.

func IsSMEAccess(accdesc : AccessDescriptor) => boolean
begin
    return accdesc.acctype == AccessType SME;
end;
```

Library pseudocode for shared/functions/memory/IsWatchpointableAccess

```
// IsWatchpointableAccess()
// =====
// Return TRUE if access should be checked for watchpoints.

func IsWatchpointableAccess(accdesc : AccessDescriptor) => boolean
begin
    return (!(accdesc.acctype IN {AccessType IFETCH,
                                   AccessType TTW,
                                   AccessType DC,
                                   AccessType IC,
                                   AccessType SPE,
                                   AccessType TRBE,
                                   AccessType AT}) ||
            (accdesc.acctype == AccessType DC &&
             accdesc.cacheop == CacheOp Invalidate &&
             (!UsingAArch32() || (ImpDefBool("DCIMVAC generates watchpoint")))));
end;
```

Library pseudocode for shared/functions/memory/MBMaintenanceScope

```
// MBMaintenanceScope
// =====
// Memory barrier scope.

type MBMaintenanceScope of enumeration {MBMaintenanceScope_Nonshareable,
                                           MBMaintenanceScope_InnerShareable,
                                           MBMaintenanceScope_OuterShareable,
                                           MBMaintenanceScope_None};
```

Library pseudocode for shared/functions/memory/MBReqTypes

```
// MBReqTypes
// =====
// Memory barrier read/write.

type MBReqTypes of enumeration {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

Library pseudocode for shared/functions/memory/MemAtomicOp

```
// MemAtomicOp
// =====
// Atomic data processing instruction types.

type MemAtomicOp of enumeration {
    MemAtomicOp_GCSSS1,
    MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP,
    MemAtomicOp_CAS,
    MemAtomicOp_FPADD,
    MemAtomicOp_FPMAX,
    MemAtomicOp_FPMIN,
    MemAtomicOp_FPMAXNM,
    MemAtomicOp_FPMINNM,
    MemAtomicOp_BFADD,
    MemAtomicOp_BFMAX,
    MemAtomicOp_BFMIN,
    MemAtomicOp_BFMAXNM,
    MemAtomicOp_BFMINNM
};
```

Library pseudocode for shared/functions/memory/MemAttrHints

```
// MemAttrHints
// =====
// Attributes and hints for Normal memory.

type MemAttrHints of record {
  attrs : bits(2), // See MemAttr_*, Cacheability attributes
  hints : bits(2), // See MemHint_*, Allocation hints
  transient : boolean
};
```

Library pseudocode for shared/functions/memory/MemOp

```
// MemOp
// =====
// Memory access instruction types.

type MemOp of enumeration {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

Library pseudocode for shared/functions/memory/MemType

```
// MemType
// =====
// Basic memory types.

type MemType of enumeration {MemType_Normal, MemType_Device};
```

Library pseudocode for shared/functions/memory/Memory

```
// Memory Tag type
// =====

type MemTagType of enumeration {
  MemTag_AllocationTagged,
  MemTag_CanonicalTagged,
  MemTag_Untagged
};
```

Library pseudocode for shared/functions/memory/MemoryAttributes

```
// MemoryAttributes
// =====
// Memory attributes descriptor

type MemoryAttributes of record {
  memtype : MemType,
  device : DeviceType, // For Device memory types
  inner : MemAttrHints, // Inner hints and attributes
  outer : MemAttrHints, // Outer hints and attributes
  shareability : Shareability, // Shareability attribute
  tags : MemTagType, // MTE tag type for this memory.
  notagaccess : boolean, // Allocation Tag access permission
  xs : bit // XS attribute
};
```

Library pseudocode for shared/functions/memory/NewAccDesc

```
// NewAccDesc()
// =====
// Create a new AccessDescriptor with initialized fields

func NewAccDesc(acctype : AccessType) => AccessDescriptor
begin
    let mpaminfo : MPAMInfo = GenMPAMCurEL(acctype);
    return NewAccDesc(acctype, mpaminfo);
end;

func NewAccDesc(acctype : AccessType, mpam : MPAMInfo) => AccessDescriptor
begin
    var accdesc : AccessDescriptor;

    accdesc.acctype           = acctype;
    accdesc.el                = PSTATE.EL;
    accdesc.ss                = SecurityStateAtEL(PSTATE.EL);
    accdesc.acqsc             = FALSE;
    accdesc.acqpc             = FALSE;
    accdesc.rels              = FALSE;
    accdesc.limitedordered    = FALSE;
    accdesc.exclusive         = FALSE;
    accdesc.rcw               = FALSE;
    accdesc.rcws              = FALSE;
    accdesc.atomicop          = FALSE;
    accdesc.nontemporal       = FALSE;
    accdesc.read              = FALSE;
    accdesc.write             = FALSE;
    accdesc.pan               = FALSE;
    accdesc.nonfault          = FALSE;
    accdesc.firstfault        = FALSE;
    accdesc.first             = FALSE;
    accdesc.contiguous        = FALSE;
    accdesc.predicated         = FALSE;
    accdesc.streamingsve      = FALSE;
    accdesc.ls64              = FALSE;
    accdesc.withstatus        = FALSE;
    accdesc.mops              = FALSE;
    accdesc.a32lsmd           = FALSE;
    accdesc.tagchecked        = FALSE;
    accdesc.tagaccess         = FALSE;
    accdesc.stzgm             = FALSE;
    accdesc.mpam              = mpam;
    accdesc.Rs                = -1;
    accdesc.Rs2               = -1;
    accdesc.Rt                = -1;
    accdesc.Rt2               = -1;
    accdesc.ispair            = FALSE;
    accdesc.highestaddressfirst = FALSE;
    accdesc.lowestaddress      = TRUE;

    return accdesc;
end;
```

Library pseudocode for shared/functions/memory/PASpace

```
// PASpace
// =====
// Physical address spaces

type PASpace of enumeration {
    PAS_Root,
    PAS_SystemAgent,
    PAS_NonSecureProtected,
    PAS_NA6,           // Reserved
    PAS_NA7,           // Reserved
    PAS_Realm,
    PAS_Secure,
    PAS_NonSecure
};
```


Library pseudocode for shared/functions/memory/Permissions

```
// Permissions
// =====
// Access Control bits in translation table descriptors

type Permissions of record {
  ap_table : bits(2), // Stage 1 hierarchical access permissions
  xn_table : bit,     // Stage 1 hierarchical execute-never for single EL regimes
  pxn_table : bit,    // Stage 1 hierarchical privileged execute-never
  uxn_table : bit,    // Stage 1 hierarchical unprivileged execute-never
  ap : bits(3),       // Stage 1 access permissions
  xn : bit,           // Stage 1 execute-never for single EL regimes
  uxn : bit,          // Stage 1 unprivileged execute-never
  pxn : bit,          // Stage 1 privileged execute-never
  ppi : bits(4),      // Stage 1 privileged indirect permissions
  upi : bits(4),      // Stage 1 unprivileged indirect permissions
  ndirty : bit,       // Stage 1 dirty state for indirect permissions scheme
  s2pi : bits(4),     // Stage 2 indirect permissions
  s2dirty : bit,      // Stage 2 dirty state
  po_index : bits(4), // Stage 1 overlay permissions index
  s2po_index : bits(4), // Stage 2 overlay permissions index
  s2ap : bits(2),     // Stage 2 access permissions
  s2tag_na : bit,     // Stage 2 tag access
  s2xnx : bit,        // Stage 2 extended execute-never
  dbm : bit,          // Dirty bit management
  s2xn : bit          // Stage 2 execute-never
};
```

Library pseudocode for shared/functions/memory/PhysMemRead

```
// PhysMemRead()
// =====
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.

impdef func PhysMemRead(size : integer{8, 16, 32, 64, 128, 256, 512}){
  desc : AddressDescriptor,
  accdesc : AccessDescriptor
  ) => (PhysMemRetStatus, bits(size))

begin
  return (ARBITRARY : PhysMemRetStatus, ARBITRARY : bits(size));
end;
```

Library pseudocode for shared/functions/memory/PhysMemRetStatus

```
// PhysMemRetStatus
// =====
// Fields that relate only to return values of PhysMem functions.

type PhysMemRetStatus of record {
  statuscode : Fault, // Fault Status
  extflag : bit,     // IMPLEMENTATION DEFINED syndrome for External aborts
  merrorstate : ErrorState, // Optional error state returned on a physical memory access
  store64bstatus : bits(64) // Status of 64B store
};
```

Library pseudocode for shared/functions/memory/PhysMemWrite

```
// PhysMemWrite()
// =====
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.

impdef func PhysMemWrite(size : integer{8, 16, 32, 64, 128, 256, 512}){
  desc : AddressDescriptor,
  accdesc : AccessDescriptor,
  value : bits(size) => PhysMemRetStatus

begin
  return ARBITRARY : PhysMemRetStatus;
end;
```

Library pseudocode for shared/functions/memory/PrefetchHint

```
// PrefetchHint
// =====
// Prefetch hint types.

type PrefetchHint of enumeration {
    Prefetch_READ,
    Prefetch_WRITE,
    Prefetch_IR,
    Prefetch_EXEC
};
```

Library pseudocode for shared/functions/memory/S1AccessControls

```
// S1AccessControls
// =====
// Effective access controls defined by stage 1 translation

type S1AccessControls of record {
    r : bit,           // Stage 1 base read permission
    w : bit,           // Stage 1 base write permission
    x : bit,           // Stage 1 base execute permission
    gcs : bit,         // Stage 1 GCS permission
    overlay : boolean, // Stage 1 FEAT_S1POE overlay applies
    or : bit,          // Stage 1 FEAT_S1POE overlay read permission
    ow : bit,          // Stage 1 FEAT_S1POE overlay write permission
    ox : bit,          // Stage 1 FEAT_S1POE overlay execute permission
    wxn : bit          // Stage 1 write permission implies execute-never
};
```

Library pseudocode for shared/functions/memory/S2AccessControls

```
// S2AccessControls
// =====
// Effective access controls defined by stage 2 translation

type S2AccessControls of record {
    r : bit,           // Stage 2 read permission.
    w : bit,           // Stage 2 write permission.
    x : bit,           // Stage 2 execute permission.
    r_rcw : bit,       // Stage 2 Read perms for RCW instruction.
    w_rcw : bit,       // Stage 2 Write perms for RCW instruction.
    r_mmu : bit,       // Stage 2 Read perms for TTW data.
    w_mmu : bit,       // Stage 2 Write perms for TTW data.
    toplevel0 : bit,   // IPA as top level table for TTBR0_EL1.
    toplevel1 : bit,   // IPA as top level table for TTBR1_EL1.
    overlay : boolean, // Overlay enable
    or : bit,          // Stage 2 overlay read permission.
    ow : bit,          // Stage 2 overlay write permission.
    ox : bit,          // Stage 2 overlay execute permission.
    or_rcw : bit,      // Stage 2 overlay Read perms for RCW instruction.
    ow_rcw : bit,      // Stage 2 overlay Write perms for RCW instruction.
    or_mmu : bit,      // Stage 2 overlay Read perms for TTW data.
    ow_mmu : bit,      // Stage 2 overlay Write perms for TTW data.
};
```

Library pseudocode for shared/functions/memory/Shareability

```
// Shareability
// =====

type Shareability of enumeration {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
// SpeculativeStoreBypassBarrierToPA()
// =====

impdef func SpeculativeStoreBypassBarrierToPA()
begin
    // Since there is no speculation in ASL model, this is a NOP.
    return;
end;
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
// SpeculativeStoreBypassBarrierToVA()
// =====

impdef func SpeculativeStoreBypassBarrierToVA()
begin
    // Since there is no speculation in ASL model, this is a NOP.
    return;
end;
```

Library pseudocode for shared/functions/memory/Tag

```
// Tag Granule size
// =====

constant LOG2_TAG_GRANULE : integer{} = 4;
constant TAG_GRANULE : integer{} = 1 << LOG2_TAG_GRANULE;
```

Library pseudocode for shared/functions/memory/VARange

```
// VARange
// =====
// Virtual address ranges

type VARange of enumeration {
    VARange_LOWER,
    VARange_UPPER
};
```

Library pseudocode for shared/functions/mpam/AltPARTIDSpace

```
// AltPARTIDSpace()
// =====
// From the Security state, EL and ALTSP configuration, determine
// whether to primary space or the alt space is selected and which
// PARTID space is the alternative space. Return that alternative
// PARTID space if selected or the primary space if not.

func AltPARTIDSpace(el : bits(2), security : SecurityState,
    primaryPIDSpace : PARTIDSpaceType) => PARTIDSpaceType
begin
    case security of
        when SS_NonSecure =>
            assert el != EL3;
            return primaryPIDSpace;
        when SS_Secure =>
            assert el != EL3;
            if primaryPIDSpace == PIDSpace_NonSecure then
                return primaryPIDSpace;
            end;
            return AltPIDSecure(el, primaryPIDSpace);
        when SS_Root =>
            assert el == EL3;
            if MPAM3_EL3().ALTSP_EL3 == '1' then
                if MPAM3_EL3().RT_ALTSP_NS == '1' then
                    return PIDSpace_NonSecure;
                else
                    return PIDSpace_Secure;
                end;
            else
                return primaryPIDSpace;
            end;
        when SS_Realm =>
            assert el != EL3;
            return AltPIDRealm(el, primaryPIDSpace);
        otherwise =>
            unreachable;
    end;
end;
```

Library pseudocode for shared/functions/mpam/AltPIDRealm

```
// AltPIDRealm()
// =====
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Realm Security state.
// Helper for AltPARTIDSpace.

func AltPIDRealm(el : bits(2), primaryPIDSpace : PARTIDSpaceType) => PARTIDSpaceType
begin
    var PIDSpace : PARTIDSpaceType = primaryPIDSpace;
    case el of
        when EL0 =>
            if ELIsInHost(EL0) then
                if !UsePrimarySpaceEL2() then
                    PIDSpace = PIDSpace NonSecure;
                end;
            elsif !UsePrimarySpaceEL10() then
                PIDSpace = PIDSpace NonSecure;
            end;
        when EL1 =>
            if !UsePrimarySpaceEL10() then
                PIDSpace = PIDSpace NonSecure;
            end;
        when EL2 =>
            if !UsePrimarySpaceEL2() then
                PIDSpace = PIDSpace NonSecure;
            end;
        otherwise =>
            unreachable;
    end;
    return PIDSpace;
end;
```

Library pseudocode for shared/functions/mpam/AltPIDSecure

```
// AltPIDSecure()
// =====
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Secure Security state.
// Helper for AltPARTIDSpace.

func AltPIDSecure(el : bits(2), primaryPIDSpace : PARTIDSpaceType) => PARTIDSpaceType
begin
    var PIDSpace : PARTIDSpaceType = primaryPIDSpace;
    case el of
        when EL0 =>
            if EL2Enabled() then
                if ELIsInHost(EL0) then
                    if !UsePrimarySpaceEL2() then
                        PIDSpace = PIDSpace NonSecure;
                    end;
                elsif !UsePrimarySpaceEL10() then
                    PIDSpace = PIDSpace NonSecure;
                end;
            elsif MPAM3_EL3().ALTSP_HEN == '0' && MPAM3_EL3().ALTSP_HFC == '1' then
                PIDSpace = PIDSpace NonSecure;
            end;
        when EL1 =>
            if EL2Enabled() then
                if !UsePrimarySpaceEL10() then
                    PIDSpace = PIDSpace NonSecure;
                end;
            elsif MPAM3_EL3().ALTSP_HEN == '0' && MPAM3_EL3().ALTSP_HFC == '1' then
                PIDSpace = PIDSpace NonSecure;
            end;
        when EL2 =>
            if !UsePrimarySpaceEL2() then
                PIDSpace = PIDSpace NonSecure;
            end;
        otherwise =>
            unreachable;
    end;
    return PIDSpace;
end;
```

Library pseudocode for shared/functions/mpam/DefaultMPAMInfo

```
// DefaultMPAMInfo()
// =====
// Returns default MPAM info. The partidspace argument sets
// the PARTID space of the default MPAM information returned.

func DefaultMPAMInfo(partidspace : PARTIDSpaceType) => MPAMInfo
begin
    var defaultinfo : MPAMInfo;
    defaultinfo.mpam_sp = partidspace;
    defaultinfo.partid = DEFAULT\_PARTID;
    defaultinfo.pmg     = DEFAULT\_PMG;
    return defaultinfo;
end;
```

Library pseudocode for shared/functions/mpam/GenMPAM

```
// GenMPAM()
// =====
// Returns MPAMInfo for Exception level el.
// If mpamdata.sm is TRUE returns MPAM information using MPAMSM_EL1.
// If mpamdata.trb is TRUE returns MPAM information using TRBMPAM_EL1.
// Otherwise returns MPAM information using MPAMn_ELx.

func GenMPAM(el_in : bits(2), mpamdata : MPAMdata, pspace : PARTIDSpaceType) => MPAMInfo
begin
    var returninfo : MPAMInfo;
    var partidel : PARTIDType;
    var perr : boolean;
    var el : bits(2) = el_in;

    // Check if the guest OS application is locked by the EL2 hypervisor to
    // only use the EL1 virtual machine's PARTIDs.
    if (el == EL0 && EL2Enabled() && MPAMHCR_EL2().GSTAPP_PLK == '1' &&
        HCR_EL2().TGE == '0') then
        el = EL1;
    end;

    (partidel, perr) = GenPARTID(el, mpamdata, pspace);
    let groupe1 : PMGType = GenPMG(el, mpamdata, perr, pspace);

    returninfo.mpam_sp = pspace;
    returninfo.partid = partidel;
    returninfo.pmg     = groupe1;
    return returninfo;
end;
```

Library pseudocode for shared/functions/mpam/GenMPAMatEL

```
// GenMPAMatEL()
// =====
// Returns MPAMinfo for the specified EL.
// May be called if MPAM is not implemented (but in a version that supports
// MPAM), MPAM is disabled, or in AArch32_ In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

func GenMPAMatEL(acctype : AccessType, el : bits(2)) => MPAMInfo
begin
    var mpamEL : bits(2);
    var pspace : PARTIDSpaceType;
    let security : SecurityState = SecurityStateAtEL(el);
    var mpamdata : MPAMdata = GenNewMPAMData();
    pspace = PARTIDSpaceFromSS(security);
    if pspace == PIDSpace NonSecure && !MPAMIsEnabled() then
        return DefaultMPAMInfo(pspace);
    end;
    mpamEL = if acctype == AccessType NV2 then EL2 else el;
    case acctype of
        when AccessType IFETCH, AccessType IC =>
            mpamdata.in_d = TRUE;
        when AccessType SME =>
            mpamdata.sm = (ImpDefBool("Shared SMCU") ||
                ImpDefBool("MPAMSM_EL1 label precedence"));
        when AccessType FP, AccessType ASIMD, AccessType SVE =>
            mpamdata.sm = (IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' &&
                (ImpDefBool("Shared SMCU") ||
                    ImpDefBool("MPAMSM_EL1 label precedence")));
        when AccessType TRBE =>
            if !SelfHostedTraceEnabled() then
                if !IsFeatureImplemented(FEAT_TRBE_MPAM) || TRBMPAM\_EL1().EN == '0' then
                    return DefaultMPAMInfo(pspace);
                else
                    mpamdata.trb = TRUE;
                end;
            end;
            if mpamdata.trb then
                var ss : SecurityState;
                case TRBMPAM\_EL1().MPAM_SP of
                    when '00' => ss = SS Secure;
                    when '01' => ss = SS NonSecure;
                    when '10' => ss = SS Root;
                    when '11' => ss = SS Realm;
                end;
                pspace = PARTIDSpaceFromSS(ss);
            end;
        otherwise =>
            // Other access types are DATA accesses
            mpamdata.in_d = FALSE;
    end;
    if IsFeatureImplemented(FEAT_RME) && MPAMIDR\_EL1().HAS_ALTSP == '1' then
        // Substitute alternative PARTID space if selected
        pspace = AltPARTIDSpace(mpamEL, security, pspace);
    end;
    if IsFeatureImplemented(FEAT_MPAMv0p1) && MPAMIDR\_EL1().HAS_FORCE_NS == '1' then
        if MPAM3\_EL3().FORCE_NS == '1' && security == SS Secure then
            pspace = PIDSpace NonSecure;
        end;
    end;
    if ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p1)) &&
        MPAMIDR\_EL1().HAS_SDEFLT == '1') then
        if MPAM3\_EL3().SDEFLT == '1' && security == SS Secure then
            return DefaultMPAMInfo(pspace);
        end;
    end;
    if !MPAMIsEnabled() then
        return DefaultMPAMInfo(pspace);
    else
        return GenMPAM(mpamEL, mpamdata, pspace);
    end;
end;
```

Library pseudocode for shared/functions/mpam/GenMPAMCurEL

```
// GenMPAMCurEL()
// =====
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in a version that supports
// MPAM), MPAM is disabled, or in AArch32_ In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

func GenMPAMCurEL(acctype : AccessType) => MPAMinfo
begin
    return GenMPAMAtEL(acctype, PSTATE.EL);
end;
```

Library pseudocode for shared/functions/mpam/GenNewMPAMData

```
// GenNewMPAMData()
// =====

func GenNewMPAMData() => MPAMdata
begin
    var mpamdata : MPAMdata;
    mpamdata.in_d = FALSE;
    mpamdata.sm = FALSE;
    mpamdata.trb = FALSE;
    return mpamdata;
end;
```

Library pseudocode for shared/functions/mpam/GenPARTID

```
// GenPARTID()
// =====
// Returns physical PARTID and error boolean for Exception level el.
// If mpamdata.sm is TRUE then PARTID is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PARTID is from TRBMPAM_EL1.
// Otherwise, the PARTID is from MPAMn_ELx.

func GenPARTID(el : bits(2), mpamdata : MPAMdata, pspace : PARTIDSpaceType) => (PARTIDType, boolean)
begin
    let partidel : PARTIDType = GetMPAM\_PARTID(el, mpamdata);
    let partid_max : PARTIDType = (if mpamdata.trb then TRBDEVID1().PARTID_MAX
                                   else MPAMIDR\_EL1().PARTID_MAX);
    if UInt(partidel) > UInt(partid_max) then
        return (DEFAULT\_PARTID, TRUE);
    end;
    if MPAMIsVirtual(el, mpamdata) then
        return MAP\_vPARTID(partidel);
    else
        return (partidel, FALSE);
    end;
end;
```

Library pseudocode for shared/functions/mpam/GenPMG

```
// GenPMG()
// =====
// Returns PMG for Exception level el.
// if mpamdata.sm is TRUE then PMG is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PMG is from TRBMPAM_EL1.
// Otherwise, PMG is from MPAMn_ELx.
// If PARTID generation (GenPARTID) encountered an error, GenPMG() should be
// called with partid_err as TRUE.

func GenPMG(el : bits(2), mpamdata : MPAMdata, partid_err : boolean,
            pspace : PARTIDSpaceType) => PMGType
begin
    if partid_err && ConstrainUnpredictableBool(Unpredictable\_USE\_DEFAULT\_PMG) then
        return DEFAULT\_PMG;
    end;
    let groupel : PMGType = GetMPAM\_PMG(el, mpamdata);
    let pmg_max : integer = (if mpamdata.trb then UInt(TRBDEVID1().PMG_MAX)
                             else UInt(MPAMIDR\_EL1().PMG_MAX));
    if UInt(groupel) <= pmg_max then
        return groupel;
    end;
    return DEFAULT\_PMG;
end;
```

Library pseudocode for shared/functions/mpam/GetMPAM_PARTID

```
// GetMPAM_PARTID()
// =====
// Returns a PARTID
// If mpamdata.sm is TRUE then PARTID is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PARTID is from TRBMPAM_EL1.
// Otherwise, the PARTID is from MPAMn_ELx.

func GetMPAM_PARTID(MPAMn : bits(2), mpamdata : MPAMdata) => PARTIDType
begin
    if mpamdata.sm then
        return MPAMSM_EL1().PARTID_D;
    end;

    if mpamdata.trb then
        return TRBMPAM_EL1().PARTID;
    end;

    if mpamdata.in_d then
        case MPAMn of
            when '11' => return MPAM3_EL3().PARTID_I;
            when '10' => return (if EL2Enabled() then MPAM2_EL2().PARTID_I else DEFAULT\_PARTID);
            when '01' => return MPAM1_EL1().PARTID_I;
            when '00' => return MPAM0_EL1().PARTID_I;
        end;
    else
        case MPAMn of
            when '11' => return MPAM3_EL3().PARTID_D;
            when '10' => return (if EL2Enabled() then MPAM2_EL2().PARTID_D else DEFAULT\_PARTID);
            when '01' => return MPAM1_EL1().PARTID_D;
            when '00' => return MPAM0_EL1().PARTID_D;
        end;
    end;
end;
```

Library pseudocode for shared/functions/mpam/GetMPAM_PMG

```
// GetMPAM_PMG()
// =====
// Returns a PMG.
// if mpamdata.sm is TRUE then PMG is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PMG is from TRBMPAM_EL1.
// Otherwise, PMG is from MPAMn_ELx.

func GetMPAM_PMG(MPAMn : bits(2), mpamdata : MPAMdata) => PMGType
begin
    if mpamdata.sm then
        return MPAMSM_EL1().PMG_D;
    end;

    if mpamdata.trb then
        return TRBMPAM_EL1().PMG;
    end;

    if mpamdata.in_d then
        case MPAMn of
            when '11' => return MPAM3_EL3().PMG_I;
            when '10' => return (if EL2Enabled() then MPAM2_EL2().PMG_I else DEFAULT\_PMG);
            when '01' => return MPAM1_EL1().PMG_I;
            when '00' => return MPAM0_EL1().PMG_I;
        end;
    else
        case MPAMn of
            when '11' => return MPAM3_EL3().PMG_D;
            when '10' => return (if EL2Enabled() then MPAM2_EL2().PMG_D else DEFAULT\_PMG);
            when '01' => return MPAM1_EL1().PMG_D;
            when '00' => return MPAM0_EL1().PMG_D;
        end;
    end;
end;
```


Library pseudocode for shared/functions/mpam/MAP_vPARTID

```
// MAP_vPARTID()
// =====
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

func MAP_vPARTID(vpartid : PARTIDType) => (PARTIDType, boolean)
begin
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    var ret : PARTIDType;
    var err : boolean;
    var virt : integer = UInt(vpartid);
    let vpmrmax : integer = UInt(MPAMIDR_EL1().VPMR_MAX);

    // vpartid_max is largest vpartid supported
    let vpartid_max : integer = (vpmrmax << 2) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if UInt(vpartid) > vpartid_max then
        virt = virt MOD (vpartid_max+1);
    end;

    // Check for valid mapping entry.
    if MPAMVPMV_EL2()[virt] == '1' then
        // vpartid has a valid mapping, access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Check for default virtual PARTID valid
    elsif MPAMVPMV_EL2()[0] == '1' then
        // Use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2()[0 +: 16];
        err = FALSE;

    // Neither is valid, use default physical PARTID.
    else
        ret = DEFAULT_PARTID;
        err = TRUE;
    end;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    let partid_max : integer = UInt(MPAMIDR_EL1().PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DEFAULT_PARTID;
        err = TRUE;
    end;
    return (ret, err);
end;
```

Library pseudocode for shared/functions/mpam/MPAM

```
// MPAM Types
// =====

type PARTIDType of bits(16);
type PMGType of bits(8);

type PARTIDSpaceType of enumeration {
    PIDSpace_Secure,
    PIDSpace_Root,
    PIDSpace_Realm,
    PIDSpace_NonSecure
};

type MPAMinfo of record {
    mpam_sp : PARTIDSpaceType,
    partid : PARTIDType,
    pmg : PMGType
};

constant DEFAULT_PARTID : PARTIDType = 0[15:0];
constant DEFAULT_PMG : PMGType = 0[7:0];

type MPAMdata of record {
    in_d : boolean, // TRUE for instruction accesses
    sm : boolean, // TRUE for SME, SVE, SIMD&FP access, and SVE prefetch
    // instructions, when the PE is in Streaming mode
    trb : boolean // TRUE for TRBE accesses using External mode when TRBMPAM_EL1.EN is 0b1
};
```

Library pseudocode for shared/functions/mpam/MPAMIsEnabled

```
// MPAMIsEnabled()
// =====
// Returns TRUE if MPAM is enabled.

func MPAMIsEnabled() => boolean
begin
    let el : bits(2) = HighestEL();
    case el of
        when EL3 => return MPAM3_EL3().MPAMEN == '1';
        when EL2 => return MPAM2_EL2().MPAMEN == '1';
        when EL1 => return MPAM1_EL1().MPAMEN == '1';
    end;
end;
```

Library pseudocode for shared/functions/mpam/MPAMIsVirtual

```
// MPAMIsVirtual()
// =====
// Returns TRUE if MPAM is configured to be virtual at the given el.

func MPAMIsVirtual(el : bits(2), mpamdata : MPAMdata) => boolean
begin
    if mpamdata.trb then
        return FALSE;
    end;

    return (MPAMIDR_EL1().HAS_HCR == '1' && EL2Enabled() &&
        ((el == EL0 && MPAMHCR_EL2().EL0_VPMEN == '1' && !ELIsInHost(EL0) ||
        (el == EL1 && MPAMHCR_EL2().EL1_VPMEN == '1')));
end;
```

Library pseudocode for shared/functions/mpam/PARTIDSpaceFromSS

```
// PARTIDSpaceFromSS()
// =====
// Returns the primary PARTID space from the Security State.

func PARTIDSpaceFromSS(security : SecurityState) => PARTIDSpaceType
begin
    case security of
        when SS NonSecure =>
            return PIDSpace NonSecure;
        when SS Root =>
            return PIDSpace Root;
        when SS Realm =>
            return PIDSpace Realm;
        when SS Secure =>
            return PIDSpace Secure;
        otherwise =>
            unreachable;
    end;
end;
```

Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL10

```
// UsePrimarySpaceEL10()
// =====
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL1 and EL0.

func UsePrimarySpaceEL10() => boolean
begin
    if MPAM3_EL3().ALTSP_HEN == '0' then
        return MPAM3_EL3().ALTSP_HFC == '0';
    end;
    return !MPAMIsEnabled() || !EL2Enabled() || MPAM2_EL2().ALTSP_HFC == '0';
end;
```

Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL2

```
// UsePrimarySpaceEL2()
// =====
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL2.

func UsePrimarySpaceEL2() => boolean
begin
    if MPAM3_EL3().ALTSP_HEN == '0' then
        return MPAM3_EL3().ALTSP_HFC == '0';
    end;
    return !MPAMIsEnabled() || MPAM2_EL2().ALTSP_EL2 == '0';
end;
```

Library pseudocode for shared/functions/mpam/mapvpmw

```
// mapvpmw()
// =====
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

func mapvpmw(vpartid : integer) => PARTIDType
begin
    var vpmw : bits(64);
    let wd : integer = vpartid DIVRM 4;
    case wd of
        when 0 => vpmw = MPAMVPM0_EL2();
        when 1 => vpmw = MPAMVPM1_EL2();
        when 2 => vpmw = MPAMVPM2_EL2();
        when 3 => vpmw = MPAMVPM3_EL2();
        when 4 => vpmw = MPAMVPM4_EL2();
        when 5 => vpmw = MPAMVPM5_EL2();
        when 6 => vpmw = MPAMVPM6_EL2();
        when 7 => vpmw = MPAMVPM7_EL2();
        otherwise => vpmw = Zeros{64};
    end;
    // vpme_lsb selects LSB of field within register
    let vpme_lsb : integer = (vpartid MOD 4) * 16;
    return vpmw[vpme_lsb +: 16];
end;
```

Library pseudocode for shared/functions/predictionrestrict/ASID

```
// ASID
// ====
// Effective ASID.

func ASID() => bits(NUM_ASIDBITS)
begin
    if ELIsInHost(EL0) then
        if TCR_EL2().A1 == '1' then
            return TTBR1_EL2().ASID;
        else
            return TTBR0_EL2().ASID;
        end;
    end;
    if !ELUsingAArch32(EL1) then
        if TCR_EL1().A1 == '1' then
            return TTBR1_EL1().ASID;
        else
            return TTBR0_EL1().ASID;
        end;
    else
        if TTBCR().EAE == '0' then
            return ZeroExtend{NUM_ASIDBITS}(CONTEXTIDR().ASID);
        else
            if TTBCR().A1 == '1' then
                return ZeroExtend{NUM_ASIDBITS}(TTBR1().ASID);
            else
                return ZeroExtend{NUM_ASIDBITS}(TTBR0().ASID);
            end;
        end;
    end;
end;
```

Library pseudocode for shared/functions/predictionrestrict/ExecutionCntxt

```
// ExecutionCntxt
// =====
// Context information for prediction restriction operation.

type ExecutionCntxt of record {
  is_vmid_valid : boolean,      // is vmid valid for current context
  all_vmid      : boolean,      // should the operation be applied for all vmids
  vmid          : bits(NUM_VMIDBITS), // if all_vmid = FALSE, vmid to which operation is applied
  is_asid_valid : boolean,      // is asid valid for current context
  all_asid      : boolean,      // should the operation be applied for all asids
  asid          : bits(NUM_ASIDBITS), // if all_asid = FALSE, ASID to which operation is applied
  target_el     : bits(2),      // target EL at which operation is performed
  security      : SecurityState,
  restriction   : RestrictType // type of restriction operation
};
```

Library pseudocode for shared/functions/predictionrestrict/RESTRICT_PREDICTIONS

```
// RESTRICT_PREDICTIONS()
// =====
// Clear all speculated values.

impdef func RESTRICT_PREDICTIONS(c : ExecutionCntxt)
begin
  return;
end;
```

Library pseudocode for shared/functions/predictionrestrict/RestrictType

```
// RestrictType
// =====
// Type of restriction on speculation.

type RestrictType of enumeration {
  RestrictType_DataValue,
  RestrictType_ControlFlow,
  RestrictType_CachePrefetch,
  RestrictType_Other          // Any other trained speculation mechanisms than those above
};
```

Library pseudocode for shared/functions/predictionrestrict/TargetSecurityState

```
// TargetSecurityState()
// =====
// Decode the target security state for the prediction context.

func TargetSecurityState(NS : bit, NSE : bit) => SecurityState
begin
  let curr_ss : SecurityState = SecurityStateAtEL(PSTATE.EL);
  if curr_ss == SS\_NonSecure then
    return SS\_NonSecure;
  elsif curr_ss == SS\_Secure then
    case NS of
      when '0' => return SS\_Secure;
      when '1' => return SS\_NonSecure;
    end;
  elsif IsFeatureImplemented(FEAT_RME) then
    if curr_ss == SS\_Root then
      case NSE::NS of
        when '00' => return SS\_Secure;
        when '01' => return SS\_NonSecure;
        when '11' => return SS\_Realm;
        when '10' => return SS\_Root;
      end;
    elsif curr_ss == SS\_Realm then
      return SS\_Realm;
    end;
  end;
  unreachable;
end;
```

Library pseudocode for shared/functions/registers/BranchTo

```
// BranchTo()
// =====
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

func BranchTo{N}(target : bits(N), branch_type : BranchType, branch_conditional : boolean)
begin
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend{64}(target);
    else
        assert N == 64 && !UsingAArch32();
        let target_vaddress : bits(64) = AArch64_BranchAddr(target[63:0], PSTATE.EL);
        if (IsFeatureImplemented(FEAT_BRBE) &&
            branch_type IN {BranchType_DIR, BranchType_INDIR,
                           BranchType_DIRCALL, BranchType_INDCALL,
                           BranchType_RET}) then
            BRBEBranch(branch_type, branch_conditional, target_vaddress);
        end;
        let branch_taken : boolean = TRUE;

        if IsFeatureImplemented(FEAT_SPE) then
            SPEBranch{N}(target, branch_type, branch_conditional, branch_taken);
        end;

        _PC = target_vaddress;
    end;
    return;
end;
```

Library pseudocode for shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// =====
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

func BranchToAddr{N}(target : bits(N), branch_type : BranchType)
begin
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend{64}(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target[63:0];
    end;
    return;
end;
```

Library pseudocode for shared/functions/registers/BranchType

```
// BranchType
// =====
// Information associated with a change in control flow.

type BranchType of enumeration {
    BranchType_DIRCALL,    // Direct Branch with link
    BranchType_INDCALL,    // Indirect Branch with link
    BranchType_ERET,       // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,        // Indirect branch with function return hint
    BranchType_DIR,        // Direct branch
    BranchType_INDIR,      // Indirect branch
    BranchType_EXCEPTION,  // Exception entry
    BranchType_RESET,      // Reset
    BranchType_UNKNOWN};   // Other
```

Library pseudocode for shared/functions/registers/Branchtypetaken

```
// Branchtypetaken
// =====
var Branchtypetaken : BranchType;
```

Library pseudocode for shared/functions/registers/ESize

```
// ESize
// =====
type ESize of integer{8, 16, 32, 64, 128};
```

Library pseudocode for shared/functions/registers/EffectiveFPCR

```
// EffectiveFPCR()
// =====
// Returns the effective FPCR value

func EffectiveFPCR() => FPCR_Type
begin
    if UsingAArch32() then
        var fpcr : FPCR_Type = ZeroExtend{64}(FPSCR());
        fpcr[7:0]   = '00000000';
        fpcr[31:27] = '00000';
        return fpcr;
    end;
    return FPCR();
end;
```

Library pseudocode for shared/functions/registers/Hint_Branch

```
// Hint_Branch()
// =====
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.

impdef func Hint_Branch(hint : BranchType)
begin
    Branchtypetaken = hint;
    return;
end;
```

Library pseudocode for shared/functions/registers/NextInstrAddr

```
// NextInstrAddr()
// =====
// Return address of the sequentially next instruction.

impdef func NextInstrAddr(N)() => bits(N)
begin
    return Zeros{N};
end;
```

Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// ResetExternalDebugRegisters()
// =====
// Reset the External Debug registers in the Core power domain.

impdef func ResetExternalDebugRegisters(cold_reset : boolean)
begin
    return;
end;
```

Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// =====
// Return address of the current instruction.

func ThisInstrAddr{N}() => bits(N)
begin
    assert N == 64 || (N == 32 && UsingAArch32());
    return PC[N-1:0];
end;
```

Library pseudocode for shared/functions/registers/UnimplementedIDRegister

```
// UnimplementedIDRegister()
// =====
// Trap access to unimplemented encodings in the feature ID register space.

func UnimplementedIDRegister()
begin
    if IsFeatureImplemented(FEAT_IDST) then
        var target_el : bits(2) = PSTATE.EL;
        if PSTATE.EL == EL0 then
            target_el = if EL2Enabled() && HCR_EL2().TGE == '1' then EL2 else EL1;
        end;
        AArch64\_SystemAccessTrap(target_el, 0x18);
    end;
    Undefined();
end;
```

Library pseudocode for shared/functions/registers/V

```
// V - accessor
// =====

accessor V{width : ESize}(n : integer) <=> value : bits(width)
begin
  // Read from SIMD&FP register with implicit slice of 8, 16
  // 32, 64 or 128 bits.
  getter
    assert n >= 0 && n <= 31;
    return _Z[[n]][width-1:0];
  end;

  // Write to SIMD&FP register with implicit extension from
  // 8, 16, 32, 64 or 128 bits.
  setter
    assert n >= 0 && n <= 31;
    if ConstrainUnpredictableBool(Unpredictable_SVEZERoupper) then
      _Z[[n]] = ZeroExtend{MAX_VL}(value);
    else
      let vlen : VecLen = if IsSVEEnabled(PSTATE.EL) then CurrentVL() else 128;
      _Z[[n]][vlen-1:0] = ZeroExtend{vlen}(value);
    end;
  end;
end;
```

Library pseudocode for shared/functions/registers/Vpart

```
// Vpart - accessor
// =====

accessor Vpart{width : ESize}(n : integer, part : integer) <=> value : bits(width)
begin
  // Reads a 128-bit SIMD&FP register in up to two parts:
  // part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
  // part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
  // value held in the register.
  getter
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
      assert width < 128;
      return V{width}(n);
    else
      assert width IN {32,64};
      let vreg : bits(128) = V{}(n);
      return vreg[(width * 2)-1:width];
    end;
  end;

  setter
    // Writes a 128-bit SIMD&FP register in up to two parts:
    // part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
    // part 1 inserts a 64-bit value into the top half of the register.
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
      assert width < 128;
      V{width}(n) = value;
    else
      assert width == 64;
      let vreg : bits(64) = V{}(n);
      V{128}(n) = value[63:0] :: vreg;
    end;
  end;
end;
```

Library pseudocode for shared/functions/registers/_PC

```
// _PC - the program counter
// =====

var _PC : bits(64);
```

Library pseudocode for shared/functions/registers/_R

```
// _R - the general-purpose register file array
// =====

var _R : array [[31]] of bits(64);
```

Library pseudocode for shared/functions/sysregisters/SPSR_ELx

```
// SPSR_ELx - accessor
// =====

accessor SPSR_ELx() <=> value : bits(64)
begin
  getter
    var result : bits(64);
    case PSTATE.EL of
      when EL1 => result = SPSR_EL1() [63:0];
      when EL2 => result = SPSR_EL2() [63:0];
      when EL3 => result = SPSR_EL3() [63:0];
      otherwise => unreachable;
    end;
    return result;
  end;

  setter
    case PSTATE.EL of
      when EL1 => SPSR_EL1() [63:0] = value[63:0];
      when EL2 => SPSR_EL2() [63:0] = value[63:0];
      when EL3 => SPSR_EL3() [63:0] = value[63:0];
      otherwise => unreachable;
    end;
  end;
end;
```

Library pseudocode for shared/functions/sysregisters/SPSR_curr

```
// SPSR_curr - accessor
// =====

accessor SPSR_curr() <=> value : bits(32)
begin
  getter
    var result : bits(32);
    case PSTATE.M of
      when M32\_FIQ => result = SPSR_fiq() [31:0];
      when M32\_IRQ => result = SPSR_irq() [31:0];
      when M32\_Svc => result = SPSR_svc() [31:0];
      when M32\_Monitor => result = SPSR_mon() [31:0];
      when M32\_Abort => result = SPSR_abt() [31:0];
      when M32\_Hyp => result = SPSR_hyp() [31:0];
      when M32\_Undef => result = SPSR_und() [31:0];
      otherwise => unreachable;
    end;
    return result;
  end;

  setter
    case PSTATE.M of
      when M32\_FIQ => SPSR_fiq() [31:0] = value[31:0];
      when M32\_IRQ => SPSR_irq() [31:0] = value[31:0];
      when M32\_Svc => SPSR_svc() [31:0] = value[31:0];
      when M32\_Monitor => SPSR_mon() [31:0] = value[31:0];
      when M32\_Abort => SPSR_abt() [31:0] = value[31:0];
      when M32\_Hyp => SPSR_hyp() [31:0] = value[31:0];
      when M32\_Undef => SPSR_und() [31:0] = value[31:0];
      otherwise => unreachable;
    end;
  end;
end;
```

Library pseudocode for shared/functions/system/AArch64_ChkFeat

```
// AArch64_ChkFeat()
// =====
// Indicates the status of some features

func AArch64_ChkFeat(feats_select : bits(64)) => bits(64)
begin
  var feat_en : bits(64) = Zeros{};
  feat_en[0] = if IsFeatureImplemented(FEAT_GCS) && GCSEnabled\(PSTATE.EL\) then '1' else '0';
  return feat_select AND NOT(feat_en);
end;
```


Library pseudocode for shared/functions/system/AddressAdd

```
// AddressAdd()
// =====
// Add an address with an offset and return the result.
// If FEAT_CPA2 is implemented, the pointer arithmetic is checked.

func AddressAdd(base : bits(64), offset : integer, accdesc : AccessDescriptor) => bits(64)
begin
    return AddressAdd(base, offset[63:0], accdesc);
end;

func AddressAdd(base : bits(64), offset : bits(64), accdesc : AccessDescriptor) => bits(64)
begin
    var result : bits(64) = base + offset;
    result = PointerAddCheckAtEL(accdesc.el, result, base);
    return result;
end;
```

Library pseudocode for shared/functions/system/AddressInNaturallyAlignedBlock

```
// AddressInNaturallyAlignedBlock()
// =====
// Returns TRUE if the given addresses are in the same naturally aligned block, and FALSE otherwise.
// An address is in a naturally aligned block if it meets any of the below conditions:
// * is a power-of-two size.
// * Is no larger than the DC ZVA block size if ESR_ELx.FnP is being set to 0b0, or EDHSR is not
//   implemented or EDHSR.FnP is being set to 0b0 (as appropriate).
// * Is no larger than the smallest implemented translation granule if ESR_ELx.FnP, or EDHSR.FnP
//   (as appropriate) is being set to 0b1.
// * Contains a watchpointed address accessed by the memory access or set of contiguous memory
//   accesses that triggered the watchpoint.

impdef func AddressInNaturallyAlignedBlock(address1 : bits(64), address2 : bits(64)) => boolean
begin
    // Returns TRUE if address in within a naturally aligned block of memory as the watchpointed
    // location, else FALSE.
    return FALSE;
end;
```

Library pseudocode for shared/functions/system/AddressIncrement

```
// AddressIncrement()
// =====
// Increment an address and return the result.
// If FEAT_CPA2 is implemented, the pointer arithmetic may be checked.

func AddressIncrement(base : bits(64), increment : integer, accdesc : AccessDescriptor) => bits(64)
begin
    return AddressIncrement(base, increment[63:0], accdesc);
end;

func AddressIncrement(base : bits(64), increment : bits(64), accdesc : AccessDescriptor) => bits(64)
begin
    var result : bits(64) = base + increment;
    // Checking the Pointer Arithmetic on an increment is equivalent to checking the
    // bytes in a sequential access crossing the 0xFFFF_FFFF_FFFF_FFFF boundary.
    if ConstrainUnpredictableBool(Unpredictable CPACHECK) then
        result = PointerAddCheckAtEL(accdesc.el, result, base);
    end;
    return result;
end;
```

Library pseudocode for shared/functions/system/BranchTargetCheck

```
// BranchTargetCheck()
// =====
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

func BranchTargetCheck()
begin
    assert IsFeatureImplemented(FEAT_BTI) && !UsingAArch32();

    // The branch target check considers the following state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if Halted() then
        return;
    elsif IsZero(PSTATE.BTYPE) then
        return;
    elsif InGuardedPage && !BTypeCompatible then
        let pc : bits(64) = ThisInstrAddr{}();
        AArch64_BranchTargetException(pc[51:0]);
    end;
end;
```

Library pseudocode for shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// =====
// Clear the Event Register of this PE.

func ClearEventRegister()
begin
    EventRegister = '0';
    return;
end;
```

Library pseudocode for shared/functions/system/ConditionHolds

```
// ConditionHolds()
// =====
// Return TRUE iff COND currently holds

func ConditionHolds(cond : bits(4)) => boolean
begin
    // Evaluate base condition.
    var result : boolean;
    case cond[3:1] of
        when '000' => result = (PSTATE.Z == '1'); // EQ or NE
        when '001' => result = (PSTATE.C == '1'); // CS or CC
        when '010' => result = (PSTATE.N == '1'); // MI or PL
        when '011' => result = (PSTATE.V == '1'); // VS or VC
        when '100' => result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
        when '101' => result = (PSTATE.N == PSTATE.V); // GE or LT
        when '110' => result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
        when '111' => result = TRUE; // AL
    end;

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond[0] == '1' && cond != '1111' then
        result = !result;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

```
// ConsumptionOfSpeculativeDataBarrier()
// =====

impdef func ConsumptionOfSpeculativeDataBarrier()
begin
    return;
end;
```

Library pseudocode for shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =====

readonly func CurrentInstrSet() => InstrSet
begin
    var result : InstrSet;
    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet A32 else InstrSet T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet A64;
    end;
    return result;
end;
```

Library pseudocode for shared/functions/system/CurrentSecurityState

```
// CurrentSecurityState()
// =====
// Returns the effective security state at the exception level based off current settings.

readonly func CurrentSecurityState() => SecurityState
begin
    return SecurityStateAtEL(PSTATE.EL);
end;
```

Library pseudocode for shared/functions/system/DSBAlias

```
// DSBAlias
// =====
// Aliases of DSB.

type DSBAlias of enumeration {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```

Library pseudocode for shared/functions/system/EL0

```
// EL0-3
// =====
// PSTATE.EL Exception level bits.

constant EL3 : bits(2) = '11';
constant EL2 : bits(2) = '10';
constant EL1 : bits(2) = '01';
constant EL0 : bits(2) = '00';
```

Library pseudocode for shared/functions/system/EL2Enabled

```
// EL2Enabled()
// =====
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Realm state when Realm EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

readonly func EL2Enabled() => boolean
begin
    var scr_ns : bit;
    if HaveEL(EL3) then
        if ELUsingAArch32(EL3) then
            scr_ns = SCR().NS;
        else
            scr_ns = SCR_EL3().NS;
        end;
    end;
    return HaveEL(EL2) && (!HaveEL(EL3) || scr_ns == '1' || IsSecureEL2Enabled());
end;
```

Library pseudocode for shared/functions/system/EL3SDDUndef

```
// EL3SDDUndef()
// =====
// Returns TRUE if in Debug state and EDSCR.SDD is set.

func EL3SDDUndef() => boolean
begin
    if Halted() && EDSCR().SDD == '1' then
        assert (PSTATE.EL != EL3 &&
            (IsFeatureImplemented(FEAT_RME) || CurrentSecurityState() != SS\_Secure));
        return TRUE;
    else
        return FALSE;
    end;
end;
```

Library pseudocode for shared/functions/system/EL3SDDUndefPriority

```
// EL3SDDUndefPriority()
// =====
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by an
// EL3 control register has priority over other traps.
// The IMPLEMENTATION DEFINED priority may be different for each case.

func EL3SDDUndefPriority() => boolean
begin
    return EL3SDDUndef() && ImpDefBool("EL3 trap priority when SDD == '1'");
end;
```

Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// =====

func ELFromM32(mode : bits(5)) => (boolean, bits(2))
begin
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid, EL):
    // 'valid' is TRUE if 'mode[4:0]' encodes a mode that is both valid for this implementation
    // and the current value of SCR.NS/SCR_EL3.NS.
    // 'EL' is the Exception level decoded from 'mode'.
    var el : bits(2);
    // Check for modes that are not valid for this implementation
    var valid : boolean = !BadMode(mode);
    let effective_nse_ns : bits(2) = EffectiveSCR\_EL3\_NSE() :: EffectiveSCR\_EL3\_NS();

    case mode of
        when M32\_Monitor =>
            el = EL3;
        when M32\_Hyp =>
            el = EL2;
        when M32\_FIQ, M32\_IRQ, M32\_Svc, M32\_Abort, M32\_Undef, M32\_System =>
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && !HaveAArch64() && SCR().NS == '0' then EL3 else EL1);
        when M32\_User =>
            el = EL0;
        otherwise =>
            valid = FALSE; // Passed an illegal mode value
    end;

    if valid && el == EL2 && HaveEL(EL3) then
        if ELUsingAArch32(EL3) && SCR().NS == '0' then
            valid = FALSE; // EL2 only valid in Non-secure state in AArch32
        elsif !ELUsingAArch32(EL3) && SCR_EL3().NS == '0' then
            valid = FALSE; // EL2 only valid in Non-secure state in AArch32
        end;
    elsif valid && IsFeatureImplemented(FEAT_RME) && effective_nse_ns == '10' then
        valid = FALSE; // Illegal Exception Return from EL3 if SCR_EL3.[NSE,NS]
        // selects a reserved encoding
    end;

    if !valid then el = ARBITRARY : bits(2); end;
    return (valid, el);
end;
```

Library pseudocode for shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// =====

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr[4:0]' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

func ELFromSPSR(N)(spsr : bits(N)) => (boolean,bits(2))
begin
  var el : bits(2);
  var valid : boolean;
  if spsr[4] == '0' then      // AArch64 state
    el = spsr[3:2];
    let effective_nse_ns : bits(2) = EffectiveSCR\_EL3\_NSE\(\) :: EffectiveSCR\_EL3\_NS\(\);
    if !HaveAArch64\(\) then
      valid = FALSE;        // No AArch64 support
    elseif !HaveEL(el) then
      valid = FALSE;        // Exception level not implemented
    elseif spsr[1] == '1' then
      valid = FALSE;        // M[1] must be 0
    elseif el == EL0 && spsr[0] == '1' then
      valid = FALSE;        // for EL0, M[0] must be 0
    elseif IsFeatureImplemented(FEAT_RME) && el != EL3 && effective_nse_ns == '10' then
      valid = FALSE;        // Only EL3 valid in Root state
    elseif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && EffectiveSCR\_EL3\_NS\(\) == '0' then
      valid = FALSE;        // Unless Secure EL2 is enabled, EL2 valid only in Non-secure state
    else
      valid = TRUE;
    end;
  elseif HaveAArch32() then  // AArch32 state
    (valid, el) = ELFromM32(spsr[4:0]);
  else
    valid = FALSE;
  end;

  if !valid then el = ARBITRARY : bits(2); end;
  return (valid,el);
end;
```

Library pseudocode for shared/functions/system/ELIsInHost

```
// ELIsInHost()
// =====

readonly func ELIsInHost(el : bits(2)) => boolean
begin
  if !IsFeatureImplemented(FEAT_VHE) || ELUsingAArch32(EL2) then
    return FALSE;
  end;
  case el of
    when EL3 =>
      return FALSE;
    when EL2 =>
      return EL2Enabled() && EffectiveHCR\_EL2\_E2H() == '1';
    when EL1 =>
      return FALSE;
    when EL0 =>
      return EL2Enabled() && EffectiveHCR\_EL2\_E2H::HCR_EL2().TGE == '11';
    otherwise =>
      unreachable;
  end;
end;
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====

readonly func ELStateUsingAArch32(el : bits(2), secure : boolean) => boolean
begin
  // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
  // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
  let (known, aarch32) : (boolean, boolean) = ELStateUsingAArch32K(el, secure);
  assert known;
  return aarch32;
end;
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// =====
// Returns (known, aarch32):
//   'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
//   using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
//   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.

readonly func ELStateUsingAArch32K(el : bits(2), secure : boolean) => (boolean, boolean)
begin
    assert HaveEL(el);

    if !HaveAArch32EL(el) then
        return (TRUE, FALSE); // Exception level is using AArch64
    elsif secure && el == EL2 then
        return (TRUE, FALSE); // Secure EL2 is using AArch64
    elsif !HaveAArch64() then
        return (TRUE, TRUE); // Highest Exception level, therefore all levels are using AArch32
    end;

    // Remainder of function deals with the interprocessing cases when highest
    // Exception level is using AArch64.

    if el == EL3 then
        return (TRUE, FALSE);
    end;

    if (HaveEL(EL3) && EffectiveSCR_EL3_RW() == '0' &&
        (!secure || !IsFeatureImplemented(FEAT_SEL2) || SCR_EL3().EEL2 == '0')) then
        // AArch32 below EL3.
        return (TRUE, TRUE);
    end;

    if el == EL2 then
        return (TRUE, FALSE);
    end;

    if (HaveEL(EL2) && !ELIsInHost(EL0) && HCR_EL2().RW == '0' &&
        (!secure || (IsFeatureImplemented(FEAT_SEL2) && SCR_EL3().EEL2 == '1')) then
        // AArch32 below EL2.
        return (TRUE, TRUE);
    end;

    if el == EL1 then
        return (TRUE, FALSE);
    end;

    // The execution state of EL0 is only known from PSTATE.nRW when executing at EL0.
    if PSTATE.EL == EL0 then
        return (TRUE, PSTATE.nRW == '1');
    else
        return (FALSE, ARBITRARY : boolean);
    end;
end;
```

Library pseudocode for shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// =====

readonly func ELUsingAArch32(el : bits(2)) => boolean
    recurselimit 1
begin
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
end;
```

Library pseudocode for shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =====

readonly func ELUsingAArch32K(el : bits(2)) => (boolean,boolean)
begin
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
end;
```

Library pseudocode for shared/functions/system/EffectiveEA

```
// EffectiveEA()
// =====
// Returns effective SCR_EL3.EA value

readonly func EffectiveEA() => bit
begin
    if !HaveEL(EL3) || Halted() then
        return '0';
    else
        return if HaveAArch64() then SCR_EL3().EA else SCR().EA;
    end;
end;
```

Library pseudocode for shared/functions/system/EffectiveHCR_EL2_E2H

```
// EffectiveHCR_EL2_E2H()
// =====
// Return the Effective HCR_EL2.E2H value.

readonly func EffectiveHCR_EL2_E2H() => bit
begin
    if !IsFeatureImplemented(FEAT_VHE) then
        return '0';
    end;

    if !IsFeatureImplemented(FEAT_E2H0) then
        return '1';
    end;

    return HCR_EL2().E2H;
end;
```

Library pseudocode for shared/functions/system/EffectiveHCR_EL2_NVx

```
// EffectiveHCR_EL2_NVx()
// =====
// Return the Effective value of HCR_EL2.[NV2,NV1,NV].

func EffectiveHCR_EL2_NVx() => bits(3)
begin
    if EL2Enabled() || !IsFeatureImplemented(FEAT_NV) then
        return '000';
    end;

    var nv1 : bit = HCR_EL2().NV1;
    if (!IsFeatureImplemented(FEAT_E2H0) &&
        ImpDefBool("HCR_EL2.NV1 is implemented as RAZ")) then
        nv1 = '0';
    end;

    if HCR_EL2().NV == '0' then
        if nv1 == '1' then
            case ConstrainUnpredictable(Unpredictable NVNV1) of
                when Constraint NVNV1_00 => return '000';
                when Constraint NVNV1_01 => return '010';
                when Constraint NVNV1_11 => return '011';
            end;
        else
            return '000';
        end;
    end;

    if !IsFeatureImplemented(FEAT_NV2) then
        return '0' :: nv1 :: '1';
    end;

    var nv2 : bit = HCR_EL2().NV2;
    if (nv2 == '0' && ImpDefBool(
        "Programming HCR_EL2.<NV,NV2> to '10' behaves as '11'")) then
        nv2 = '1';
    end;

    return nv2 :: nv1 :: '1';
end;
```

Library pseudocode for shared/functions/system/EffectiveSCR_EL3_FIQ

```
// EffectiveSCR_EL3_FIQ()
// =====
// Returns effective SCR_EL3().FIQ value

readonly func EffectiveSCR_EL3_FIQ() => bit
begin
    if !HaveEL(EL3) || ELUsingAArch32(EL3) then
        return '0';
    end;
    return SCR_EL3().FIQ;
end;
```

Library pseudocode for shared/functions/system/EffectiveSCR_EL3_IRQ

```
// EffectiveSCR_EL3_IRQ()
// =====
// Returns effective SCR_EL3().IRQ value

readonly func EffectiveSCR_EL3_IRQ() => bit
begin
    if !HaveEL(EL3) || ELUsingAArch32(EL3) then
        return '0';
    end;
    return SCR_EL3().IRQ;
end;
```

Library pseudocode for shared/functions/system/EffectiveSCR_EL3_NS

```
// EffectiveSCR_EL3_NS()
// =====
// Return Effective SCR_EL3.NS value.

readonly func EffectiveSCR_EL3_NS() => bit
begin
    if !HaveSecureState() then
        return '1';
    elsif !HaveEL(EL3) then
        return '0';
    elsif ELUsingAArch32(EL3) then
        return SCR().NS;
    else
        return SCR_EL3().NS;
    end;
end;
```

Library pseudocode for shared/functions/system/EffectiveSCR_EL3_NSE

```
// EffectiveSCR_EL3_NSE()
// =====
// Return Effective SCR_EL3.NSE value.

readonly func EffectiveSCR_EL3_NSE() => bit
begin
    return if !IsFeatureImplemented(FEAT_RME) then '0' else SCR_EL3().NSE;
end;
```

Library pseudocode for shared/functions/system/EffectiveSCR_EL3_RW

```
// EffectiveSCR_EL3_RW()
// =====
// Returns effective SCR_EL3.RW value

readonly func EffectiveSCR_EL3_RW() => bit
begin
    if !HaveAArch64() then
        return '0';
    end;
    if !HaveAArch32EL(EL2) && !HaveAArch32EL(EL1) then
        return '1';
    end;
    if HaveAArch32EL(EL1) then
        if !HaveAArch32EL(EL2) && EffectiveSCR_EL3_NS() == '1' then
            return '1';
        end;
        if (IsFeatureImplemented(FEAT_SEL2) && SCR_EL3().EEL2 == '1' &&
            EffectiveSCR_EL3_NS() == '0') then
            return '1';
        end;
    end;
    return SCR_EL3().RW;
end;
```


Library pseudocode for shared/functions/system/EffectiveTGE

```
// EffectiveTGE()
// =====
// Returns effective TGE value

func EffectiveTGE() => bit
begin
  if EL2Enabled() then
    return if ELUsingAArch32\(EL2\) then HCR().TGE else HCR_EL2().TGE;
  else
    return '0';          // Effective value of TGE is zero
  end;
end;
```

Library pseudocode for shared/functions/system/EndOfInstruction

```
// EndOfInstruction()
// =====
// Terminate processing of the current instruction.

noreturn impdef func EndOfInstruction(EndReason : EndOfInstructionReason)
begin
  throw EndOfInstructionException{reason=EndReason};
end;
```

Library pseudocode for shared/functions/system/EndOfInstructionException

```
// EndOfInstructionException
// =====
// End of Instruction exception type.

type EndOfInstructionException of exception {
  reason : EndOfInstructionReason
};
```

Library pseudocode for shared/functions/system/EndOfInstructionReason

```
// EndOfInstructionReason
// =====

type EndOfInstructionReason of enumeration {
  EndOfInstructionReason_Retired,
  EndOfInstructionReason_Exception
};
```

Library pseudocode for shared/functions/system/EnterLowPowerState

```
// EnterLowPowerState()
// =====
// PE enters a low-power state.

impdef func EnterLowPowerState()
begin
  return;
end;
```

Library pseudocode for shared/functions/system/EventRegister

```
// EventRegister
// =====
// Event Register for this PE.

var EventRegister : bits(1);
```

Library pseudocode for shared/functions/system/ExceptionalOccurrenceTargetState

```
// ExceptionalOccurrenceTargetState
// =====
// Enumeration to represent the target state of an Exceptional Occurrence.
// The Exceptional Occurrence can be either Exception or Debug State entry.

type ExceptionalOccurrenceTargetState of enumeration {
  AArch32_NonDebugState,
  AArch64_NonDebugState,
  DebugState
};
```

Library pseudocode for shared/functions/system/ExecuteAsNOP

```
// ExecuteAsNOP()
// =====

func ExecuteAsNOP()
begin
    EndOfInstruction(EndOfInstructionReason\_Retired);
end;
```

Library pseudocode for shared/functions/system/FIQPending

```
// FIQPending()
// =====
// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.

impdef func FIQPending() => (boolean, boolean)
begin
    return (FALSE, FALSE);
end;
```

Library pseudocode for shared/functions/system/GetAccumulatedFPEExceptions

```
// GetAccumulatedFPEExceptions()
// =====
// Returns FP exceptions accumulated by the PE.

impdef func GetAccumulatedFPEExceptions() => bits(8)
begin
    return ARBITRARY : bits(8);
end;
```

Library pseudocode for shared/functions/system/GetLoadStoreType

```
// GetLoadStoreType()
// =====
// Returns the Load/Store Type. Used when a Translation fault,
// Access flag fault, or Permission fault generates a Data Abort.

impdef func GetLoadStoreType() => bits(2)
begin
    return ARBITRARY : bits(2);
end;
```

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE

func GetPSRFromPSTATE{N}(targetELState : ExceptionalOccurrenceTargetState) => bits(N)
begin
    if UsingAArch32() && targetELState == AArch32\_NonDebugState then
        assert N == 32;
    else
        assert N == 64;
    end;

    var spsr : bits(N) = Zeros{};
    if IsFeatureImplemented(FEAT_UINJ) && targetELState == DebugState then
        spsr[36] = PSTATE.UINJ;
    end;
    spsr[31:28] = PSTATE.[N,Z,C,V];
    if IsFeatureImplemented(FEAT_PAN) then spsr[22] = PSTATE.PAN; end;
    spsr[20] = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr[27] = PSTATE.Q;
        spsr[26:25] = PSTATE.IT[1:0];
        if IsFeatureImplemented(FEAT_SSBS) then spsr[23] = PSTATE.SSBS; end;
        if IsFeatureImplemented(FEAT_DIT) then
            if targetELState == AArch32\_NonDebugState then
                spsr[21] = PSTATE.DIT;
            else // AArch64_NonDebugState or DebugState
                spsr[24] = PSTATE.DIT;
            end;
        end;
        if targetELState IN {AArch64\_NonDebugState, DebugState} then
            spsr[21] = PSTATE.SS;
        end;
        spsr[19:16] = PSTATE.GE;
        spsr[15:10] = PSTATE.IT[7:2];
        spsr[9] = PSTATE.E;
        spsr[8:6] = PSTATE.[A,I,F]; // No PSTATE.D in AArch32 state
        spsr[5] = PSTATE.T;
        assert PSTATE.M[4] == PSTATE.nRW; // bit [4] is the discriminator
        spsr[4:0] = PSTATE.M;
    else // AArch64 state
        if IsFeatureImplemented(FEAT_PAuth_LR) then spsr[35] = PSTATE.PACM; end;
        if IsFeatureImplemented(FEAT_GCS) then spsr[34] = PSTATE.EXLOCK; end;
        if (IsFeatureImplemented(FEAT_EBEP) || IsFeatureImplemented(FEAT_SPE_EXC) ||
            IsFeatureImplemented(FEAT_TRBE_EXC)) then
            spsr[32] = PSTATE.PM;
        end;
        if IsFeatureImplemented(FEAT_MTE) then spsr[25] = PSTATE.TCO; end;
        if IsFeatureImplemented(FEAT_DIT) then spsr[24] = PSTATE.DIT; end;
        if IsFeatureImplemented(FEAT_UAO) then spsr[23] = PSTATE.UAO; end;
        spsr[21] = PSTATE.SS;
        if IsFeatureImplemented(FEAT_NMI) then spsr[13] = PSTATE.ALLINT; end;
        if IsFeatureImplemented(FEAT_SSBS) then spsr[12] = PSTATE.SSBS; end;
        if IsFeatureImplemented(FEAT_BTI) then spsr[11:10] = PSTATE.BTTYPE; end;
        spsr[9:6] = PSTATE.[D,A,I,F];
        spsr[4] = PSTATE.nRW;
        spsr[3:2] = PSTATE.EL;
        spsr[0] = PSTATE.SP;
    end;
    return spsr;
end;
```

Library pseudocode for shared/functions/system/HaveAArch32

```
// HaveAArch32()
// =====
// Return TRUE if AArch32 state is supported at at least EL0.

readonly func HaveAArch32() => boolean
begin
    return IsFeatureImplemented(FEAT_AA32EL0);
end;
```

Library pseudocode for shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// =====
// Return TRUE if Exception level 'el' supports AArch32 in this implementation

readonly func HaveAArch32EL(el : bits(2)) => boolean
begin
    case el of
        when EL0 => return IsFeatureImplemented(FEAT_AA32EL0);
        when EL1 => return IsFeatureImplemented(FEAT_AA32EL1);
        when EL2 => return IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 => return IsFeatureImplemented(FEAT_AA32EL3);
    end;
end;
```

Library pseudocode for shared/functions/system/HaveAArch64

```
// HaveAArch64()
// =====
// Return TRUE if the highest Exception level is using AArch64 state.

readonly func HaveAArch64() => boolean
begin
    return (IsFeatureImplemented(FEAT_AA64EL0)
        || IsFeatureImplemented(FEAT_AA64EL1)
        || IsFeatureImplemented(FEAT_AA64EL2)
        || IsFeatureImplemented(FEAT_AA64EL3)
    );
end;
```

Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

readonly func HaveEL(el : bits(2)) => boolean
begin
    case el of
        when EL1, EL0 =>
            return TRUE; // EL1 and EL0 must exist
        when EL2 =>
            return IsFeatureImplemented(FEAT_AA64EL2) || IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 =>
            return IsFeatureImplemented(FEAT_AA64EL3) || IsFeatureImplemented(FEAT_AA32EL3);
        otherwise =>
            unreachable;
    end;
end;
```

Library pseudocode for shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// =====
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

readonly func HaveELUsingSecurityState(el : bits(2), secure : boolean) => boolean
begin
    case el of
        when EL3 =>
            assert secure;
            return HaveEL(EL3);
        when EL2 =>
            if secure then
                return HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2);
            else
                return HaveEL(EL2);
            end;
        otherwise =>
            return (HaveEL(EL3) ||
                (secure == ImpDefBool("Secure-only implementation")));
    end;
end;
```

Library pseudocode for shared/functions/system/HaveSecureState

```
// HaveSecureState()
// =====
// Return TRUE if Secure State is supported.

readonly func HaveSecureState() => boolean
begin
    if !HaveEL(EL3) then
        return SecureOnlyImplementation();
    end;
    if IsFeatureImplemented(FEAT_RME) && !IsFeatureImplemented(FEAT_SEL2) then
        return FALSE;
    end;
    return TRUE;
end;
```

Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// =====
// Returns the highest implemented Exception level.

readonly func HighestEL() => bits(2)
begin
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
    end;
end;
```

Library pseudocode for shared/functions/system/Hint_CLRBHB

```
// Hint_CLRBHB()
// =====
// Provides a hint to clear the branch history for the current context.

impdef func Hint_CLRBHB()
begin
    return;
end;
```

Library pseudocode for shared/functions/system/Hint_DGH

```
// Hint_DGH()
// =====
// Provides a hint to close any gathering occurring within the implementation.

impdef func Hint_DGH()
begin
    return;
end;
```

Library pseudocode for shared/functions/system/Hint_StoreShared

```
// Hint_StoreShared()
// =====
// Provides a hint that if the next instruction is an explicit write it is being waited on by
// observers and as such the data should propagate to them with minimum latency.
// A stream value of FALSE indicates KEEP whilst a value of TRUE indicates STRM.

impdef func Hint_StoreShared(stream : boolean)
begin
    return;
end;
```

Library pseudocode for shared/functions/system/Hint_WFE

```
// Hint_WFE()
// =====
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs.

func Hint_WFE()
begin
    if IsEventRegisterSet() then
        ClearEventRegister();
        return;
    end;

    var trap : boolean;
    var target_el : bits(2);
    (trap, target_el) = AArch64\_CheckForWfxTrap(WfxType_WFE);
    if trap then
        if IsFeatureImplemented(FEAT_TWED) then
            // Determine if trap delay is enabled and delay amount
            var delay_enabled : boolean;
            var delay : integer;
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if WaitForEventUntilDelay(delay_enabled, delay) then
                // Event arrived before delay
                return;
            end;
        end;

        // Proceed with trapping
        if target_el == EL3 && EL3SDDUndef() then
            Undefined();
        else
            AArch64\_WfxTrap(WfxType_WFE, target_el);
        end;
    else
        WaitForEvent();
    end;
end;
```

Library pseudocode for shared/functions/system/Hint_WFET

```
// Hint_WFET()
// =====
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET, a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

func Hint_WFET(localtimeout : integer)
begin
    if IsEventRegisterSet() then
        ClearEventRegister();
        return;
    end;

    if IsFeatureImplemented(FEAT_WFXT) && LocalTimeoutEvent(localtimeout) then
        // No further operation if the local timeout has expired.
        EndOfInstruction(EndOfInstructionReason\_Retired);
    end;

    var trap : boolean;
    var target_el : bits(2);
    (trap, target_el) = AArch64\_CheckForWfxTrap(WfxType_WFET);
    if trap then
        if IsFeatureImplemented(FEAT_TWED) then
            // Determine if trap delay is enabled and delay amount
            var delay_enabled : boolean;
            var delay : integer;
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if WaitForEventUntilDelay(delay_enabled, delay) then
                // Event arrived before the delay expired
                return;
            end;
        end;

        // Proceed with trapping
        if target_el == EL3 && EL3SDDUndef() then
            Undefined();
        else
            AArch64\_WfxTrap(WfxType_WFET, target_el);
        end;
    else
        WaitForEvent(localtimeout);
    end;
end;
```

Library pseudocode for shared/functions/system/Hint_WFI

```
// Hint_WFI()
// =====
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs.

func Hint_WFI()
begin
  if AArch64 InterruptPending\(\) then
    // No further operation if an interrupt is pending.
    EndOfInstruction(EndOfInstructionReason Retired);
  end;

  var trap : boolean;
  var target_el : bits(2);
  (trap, target_el) = AArch64 CheckForWfxTrap\(WFxType WFI\);
  if trap then
    if target_el == EL3 && EL3SDDUndef\(\) then
      Undefined();
    end;
    AArch64 WfxTrap\(WFxType WFI, target\_el\);
  else
    WaitForInterrupt\(\);
  end;
end;
```

Library pseudocode for shared/functions/system/Hint_WFIT

```
// Hint_WFIT()
// =====
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

func Hint_WFIT(localtimeout : integer)
begin
  if (AArch64 InterruptPending\(\) || (IsFeatureImplemented(FEAT_WFXT) &&
    LocalTimeoutEvent\(localtimeout\))) then
    // No further operation if an interrupt is pending or the local timeout has expired.
    EndOfInstruction(EndOfInstructionReason Retired);
  end;

  var trap : boolean;
  var target_el : bits(2);
  (trap, target_el) = AArch64 CheckForWfxTrap\(WFxType WFIT\);
  if trap then
    if target_el == EL3 && EL3SDDUndef\(\) then
      Undefined();
    end;
    AArch64 WfxTrap\(WFxType WFIT, target\_el\);
  else
    WaitForInterrupt\(localtimeout\);
  end;
end;
```

Library pseudocode for shared/functions/system/Hint_Yield

```
// Hint_Yield()
// =====
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.

impdef func Hint_Yield()
begin
  return;
end;
```

Library pseudocode for shared/functions/system/IRQPending

```
// IRQPending()
// =====
// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.

impdef func IRQPending() => (boolean, boolean)
begin
  return (FALSE, FALSE);
end;
```

Library pseudocode for shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// =====

func IllegalExceptionReturn{N}(spsr : bits(N)) => boolean
begin
    // Check for illegal return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state, when SecureEL2 is not enabled.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    let (valid, target) : (boolean, bits(2)) = ELFromSPSR{N}(spsr);
    if !valid then return TRUE; end;

    // Check for return to higher Exception level.
    if UInt(target) > UInt(PSTATE.EL) then return TRUE; end;

    let spsr_mode_is_aarch32 : boolean = (spsr[4] == '1');

    // Check for illegal return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //   Execution state used in the Exception level being returned to, as determined by
    //   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    // * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    // * To AArch64 state from AArch32 state (should be caught by above).
    let (known, target_el_is_aarch32) : (boolean, boolean) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE; end;

    // Check for illegal return from AArch32 to AArch64.
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE; end;
    // Check for illegal return to EL1 when HCR_EL2.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if EL2Enabled() && target == EL1 && HCR_EL2().TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE; end;
    end;

    if (IsFeatureImplemented(FEAT_GCS) && PSTATE.EXLOCK == '0' &&
        PSTATE.EL == target && GetCurrentEXLOCKEN()) then
        return TRUE;
    end;

    return FALSE;
end;
```

Library pseudocode for shared/functions/system/InstrSet

```
// InstrSet
// =====

type InstrSet of enumeration {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

Library pseudocode for shared/functions/system/InstructionFetchBarrier

```
// InstructionFetchBarrier()
// =====

impdef func InstructionFetchBarrier()
begin
    return;
end;
```

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// =====

impdef func InstructionSynchronizationBarrier()
begin
    return;
end;
```


Library pseudocode for shared/functions/system/IsASEInstruction

```
// IsASEInstruction()
// =====
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

impdef func IsASEInstruction() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/system/IsCurrentSecurityState

```
// IsCurrentSecurityState()
// =====
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

func IsCurrentSecurityState(ss : SecurityState) => boolean
begin
    return CurrentSecurityState() == ss;
end;
```

Library pseudocode for shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// =====
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

func IsEventRegisterSet() => boolean
begin
    return EventRegister == '1';
end;
```

Library pseudocode for shared/functions/system/IsHighestEL

```
// IsHighestEL()
// =====
// Returns TRUE if given exception level is the highest exception level implemented

readonly func IsHighestEL(el : bits(2)) => boolean
begin
    return HighestEL() == el;
end;
```

Library pseudocode for shared/functions/system/IsInHost

```
// IsInHost()
// =====

readonly func IsInHost() => boolean
begin
    return ELIsInHost(PSTATE.EL);
end;
```

Library pseudocode for shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// =====
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

readonly func IsSecureBelowEL3() => boolean
begin
    if HaveEL(EL3) then
        return if !HaveAArch64() then SCR().NS == '0' else SCR_EL3().NS == '0';
    elsif HaveEL(EL2) && (!IsFeatureImplemented(FEAT_SEL2) || !HaveAArch64()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if PE is Secure or FALSE if Non-secure.
        return ImpDefBool("Secure-only implementation");
    end;
end;
```

Library pseudocode for shared/functions/system/IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// =====
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

readonly func IsSecureEL2Enabled() => boolean
begin
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) then
        if HaveEL(EL3) then
            if !ELUsingAArch32(EL3) && SCR_EL3().EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
            end;
        else
            return SecureOnlyImplementation();
        end;
    else
        return FALSE;
    end;
end;
```

Library pseudocode for shared/functions/system/LocalTimeoutEvent

```
// LocalTimeoutEvent()
// =====
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.

func LocalTimeoutEvent(localtimeout : integer) => boolean
begin
    assert localtimeout >= 0;

    let cntvct : bits(64) = VirtualCounterTimer();
    if UInt(cntvct) >= localtimeout then
        return TRUE;
    end;

    IsLocalTimeoutEventPending = TRUE;
    LocalTimeoutVal = localtimeout[63:0]; // Store value to compare against
                                           // Virtual Counter Timer at subsequent clock ticks

    return FALSE;
end;
```

Library pseudocode for shared/functions/system/Mode

```
// Mode bits
// =====
// AArch32 PSTATE.M mode bits.

constant M32_User : bits(5)    = '10000';
constant M32_FIQ : bits(5)     = '10001';
constant M32_IRQ : bits(5)     = '10010';
constant M32_Svc : bits(5)     = '10011';
constant M32_Monitor : bits(5) = '10110';
constant M32_Abort : bits(5)   = '10111';
constant M32_Hyp : bits(5)     = '11010';
constant M32_Undef : bits(5)   = '11011';
constant M32_System : bits(5)  = '11111';
```

Library pseudocode for shared/functions/system/NonSecureOnlyImplementation

```
// NonSecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Non-secure for this implementation.

func NonSecureOnlyImplementation() => boolean
begin
    return ImpDefBool("Non-secure only implementation");
end;
```

Library pseudocode for shared/functions/system/PLOfEL

```
// PLOfEL()
// =====

func PLOfEL(e1 : bits(2)) => PrivilegeLevel
begin
  case e1 of
    when EL3 => return if !HaveAArch64() then PL1 else PL3;
    when EL2 => return PL2;
    when EL1 => return PL1;
    when EL0 => return PL0;
  end;
end;
```

Library pseudocode for shared/functions/system/PSTATE

```
// PSTATE
// =====
// PE state bits.
// There is no significance to the field order.

var PSTATE : collection {
  N : bits (1),          // Negative condition flag
  Z : bits (1),          // Zero condition flag
  C : bits (1),          // Carry condition flag
  V : bits (1),          // Overflow condition flag
  D : bits (1),          // Debug mask bit [AArch64 only]
  A : bits (1),          // SError interrupt mask bit
  I : bits (1),          // IRQ mask bit
  F : bits (1),          // FIQ mask bit
  EXLOCK : bits (1),     // Lock exception return state
  PAN : bits (1),        // Privileged Access Never Bit [v8.1]
  UAO : bits (1),        // User Access Override [v8.2]
  DIT : bits (1),        // Data Independent Timing [v8.4]
  TCO : bits (1),        // Tag Check Override [v8.5, AArch64 only]
  PM : bits (1),         // PMU exception Mask
  BTYPE : bits (2),      // Branch Type [v8.5]
  PACM : bits (1),       // PAC instruction modifier
  ZA : bits (1),         // Accumulation array enabled [SME]
  SM : bits (1),         // Streaming SVE mode enabled [SME]
  ALLINT : bits (1),     // Interrupt mask bit
  UINJ : bits (1),       // Undefined Exception Injection
  SS : bits (1),         // Software step bit
  IL : bits (1),         // Illegal Execution state bit
  EL : bits (2),         // Exception level
  nRW : bits (1),        // Execution state: 0=AArch64, 1=AArch32
  SP : bits (1),         // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
  Q : bits (1),          // Cumulative saturation flag [AArch32 only]
  GE : bits (4),         // Greater than or Equal flags [AArch32 only]
  SSBS : bits (1),       // Speculative Store Bypass Safe
  IT : bits (8),         // If-then bits, RES0 in CPSR [AArch32 only]
  J : bits (1),          // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]
  T : bits (1),          // T32 bit, RES0 in CPSR [AArch32 only]
  E : bits (1),          // Endianness bit [AArch32 only]
  M : bits (5),          // Mode field [AArch32 only]
};
```

Library pseudocode for shared/functions/system/PhysicalCountInt

```
// PhysicalCountInt()
// =====
// Returns the integral part of physical count value of the System counter.

func PhysicalCountInt() => bits(64)
begin
  return PhysicalCount[87:24];
end;
```

Library pseudocode for shared/functions/system/PrivilegeLevel

```
// PrivilegeLevel
// =====
// Privilege Level abstraction.

type PrivilegeLevel of enumeration {PL3, PL2, PL1, PL0};
```

Library pseudocode for shared/functions/system/RestoredITBits

```
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

func RestoredITBits(N) (spsr : bits(N)) => bits(8)
begin
    let it : bits(8) = spsr[15:10,26:25];

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;
    end;
end;

// The IT bits are forced to zero when they are set to a reserved value.
if !IsZero(it[7:4]) && IsZero(it[3:0]) then
    return '00000000';
end;

// The IT bits are forced to zero when returning to A32 state, or when returning to an EL
// with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
let itd : bit = if PSTATE.EL == EL2 then HSCTLR().ITD else SCTLR().ITD;
if (spsr[5] == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it[2:0])) then
    return '00000000';
else
    return it;
end;
end;
```

Library pseudocode for shared/functions/system/SecureOnlyImplementation

```
// SecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Secure for this implementation.

readonly func SecureOnlyImplementation() => boolean
begin
    return ImpDefBool("Secure-only implementation");
end;
```

Library pseudocode for shared/functions/system/SecurityState

```
// SecurityState
// =====
// The Security state of an execution context

type SecurityState of enumeration {
    SS_NonSecure,
    SS_Root,
    SS_Realm,
    SS_Secure
};
```

Library pseudocode for shared/functions/system/SecurityStateAtEL

```
// SecurityStateAtEL()
// =====
// Returns the effective security state at the exception level based off current settings.

readonly func SecurityStateAtEL(EL : bits(2)) => SecurityState
begin
    if IsFeatureImplemented(FEAT_RME) then
        if EL == EL3 then return SS Root; end;
        let effective_nse_ns : bits(2) = EffectiveSCR\_EL3\_NSE() :: EffectiveSCR\_EL3\_NS();
        case effective_nse_ns of
            when '00' =>
                if IsFeatureImplemented(FEAT_SEL2) then
                    return SS Secure;
                else
                    unreachable;
                end;
            when '01' =>
                return SS NonSecure;
            when '11' =>
                return SS Realm;
            otherwise =>
                unreachable;
        end;
    end;

    if !HaveEL(EL3) then
        if SecureOnlyImplementation() then
            return SS Secure;
        else
            return SS NonSecure;
        end;
    elsif EL == EL3 then
        return SS Secure;
    else
        // For EL2 call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled());
        if !ELUsingAArch32(EL3) then
            return if EffectiveSCR\_EL3\_NS() == '1' then SS NonSecure else SS Secure;
        else
            return if SCR().NS == '1' then SS NonSecure else SS Secure;
        end;
    end;
end;
```

Library pseudocode for shared/functions/system/SendEvent

```
// SendEvent()
// =====
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.

impdef func SendEvent()
begin
    SendEventLocal();
end;
```

Library pseudocode for shared/functions/system/SendEventLocal

```
// SendEventLocal()
// =====
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

func SendEventLocal()
begin
    EventRegister = '1';
    return;
end;
```

Library pseudocode for shared/functions/system/SetAccumulatedFPExceptions

```
// SetAccumulatedFPExceptions()
// =====
// Stores FP Exceptions accumulated by the PE.

impdef func SetAccumulatedFPExceptions(accumulated_exceptions : bits(8))
begin
    return;
end;
```

Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// =====

func SetPSTATEFromPSR{N}(spsr : bits(N))
begin
    let illegal_psr_state : boolean = IllegalExceptionReturn{N}(spsr);
    SetPSTATEFromPSR{N}(spsr, illegal_psr_state);
end;

// SetPSTATEFromPSR()
// =====
// Set PSTATE based on a PSR value

func SetPSTATEFromPSR{N}(spsr_in : bits(N), illegal_psr_state : boolean)
begin
    var spsr : bits(N) = spsr_in;
    let from_aarch64 : boolean = !UsingAArch32();
    PSTATE.SS = DebugExceptionReturnSS{N}(spsr);
    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTTYPE = ARBITRARY : bits(2); end;
        if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = ARBITRARY : bit; end;
        if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0'; end;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr[20];
        if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = spsr[36]; end;
        if spsr[4] == '1' then // AArch32 state
            AArch32\_WriteMode(spsr[4:0]); // Sets PSTATE.EL correctly
            if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = spsr[23]; end;
        else // AArch64 state
            PSTATE.nRW = '0';
            (-, PSTATE.EL) = ELFromSPSR{N}(spsr);
            PSTATE.SP = spsr[0];
            if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTTYPE = spsr[11:10]; end;
            if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = spsr[12]; end;
            if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = spsr[23]; end;
            if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = spsr[24]; end;
            if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = spsr[25]; end;
            if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = spsr[34]; end;
            if IsFeatureImplemented(FEAT_PAuth_LR) then
                PSTATE.PACM = if IsPACMEnabled() then spsr[35] else '0';
            end;
        end;
    end;
end;

// If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
// copied from SPSR.
if PSTATE.IL == '1' && PSTATE.nRW == '1' then
    if ConstrainUnpredictableBool(Unpredictable\_ILZerot) then spsr[5] = '0'; end;
end;

// State that is reinstated regardless of illegal exception return
PSTATE.[N,Z,C,V] = spsr[31:28];
if IsFeatureImplemented(FEAT_PAN) then PSTATE.PAN = spsr[22]; end;
if PSTATE.nRW == '1' then // AArch32 state
    PSTATE.Q = spsr[27];
    PSTATE.IT = RestoredITBits{N}(spsr);
    ShouldAdvanceIT = FALSE;
    if IsFeatureImplemented(FEAT_DIT) then
        PSTATE.DIT = (if (Restarting()) || from_aarch64) then spsr[24] else spsr[21]);
    end;
    PSTATE.GE = spsr[19:16];
    PSTATE.E = spsr[9];
    PSTATE.[A,I,F] = spsr[8:6]; // No PSTATE.D in AArch32 state
    PSTATE.T = spsr[5]; // PSTATE.J is RES0
else // AArch64 state
    if (IsFeatureImplemented(FEAT_EBEP) || IsFeatureImplemented(FEAT_SPE_EXC) ||
        IsFeatureImplemented(FEAT_TRBE_EXC)) then
        PSTATE.PM = spsr[32];
    end;
    if IsFeatureImplemented(FEAT_NMI) then PSTATE.ALLINT = spsr[13]; end;
    PSTATE.[D,A,I,F] = spsr[9:6]; // No PSTATE.[Q,IT,GE,E,T] in AArch64 state
end;
return;
end;
```

Library pseudocode for shared/functions/system/ShouldAdvanceHS

```
// ShouldAdvanceHS
// =====
// Cleared if we should not advance the EDESR.SS after the current instruction.

var ShouldAdvanceHS : boolean;
```

Library pseudocode for shared/functions/system/ShouldAdvanceIT

```
// ShouldAdvanceIT
// =====
// Cleared if we should not advance the PSTATE.IT after the current instruction.

var ShouldAdvanceIT : boolean;
```

Library pseudocode for shared/functions/system/ShouldAdvanceSS

```
// ShouldAdvanceSS
// =====
// Cleared if PSTATE.SS is written by the current instruction.

var ShouldAdvanceSS : boolean;
```

Library pseudocode for shared/functions/system/SmallestTranslationGranule

```
// SmallestTranslationGranule()
// =====
// Smallest implemented translation granule.

func SmallestTranslationGranule() => integer
begin
    if IsFeatureImplemented(FEAT_TGran4K) then return 12; end;
    if IsFeatureImplemented(FEAT_TGran16K) then return 14; end;
    if IsFeatureImplemented(FEAT_TGran64K) then return 16; end;
    unreachable;
end;
```

Library pseudocode for shared/functions/system/SpeculationBarrier

```
// SpeculationBarrier()
// =====

impdef func SpeculationBarrier()
begin
    // Since there is no speculation in asl model, this is a NOP.
    return;
end;
```

Library pseudocode for shared/functions/system/SynchronizeContext

```
// SynchronizeContext()
// =====
// Context Synchronization event, includes Instruction Fetch Barrier effect

impdef func SynchronizeContext()
begin
    InstructionFetchBarrier();
end;
```

Library pseudocode for shared/functions/system/SynchronizeErrors

```
// SynchronizeErrors()
// =====
// Implements the error synchronization event.

impdef func SynchronizeErrors()
begin
    return;
end;
```

Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// TakeUnmaskedPhysicalSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt.

impdef func TakeUnmaskedPhysicalSErrorInterrupts(iesb_req : boolean)
begin
    return;
end;
```

Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// TakeUnmaskedSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.

impdef func TakeUnmaskedSErrorInterrupts()
begin
    return;
end;
```

Library pseudocode for shared/functions/system/ThisInstr

```
// ThisInstr()
// =====
// Return the bitstring encoding of the currently executing instruction.

impdef func ThisInstr() => bits(32)
begin
    Unimplemented();
end;
```

Library pseudocode for shared/functions/system/ThisInstrLength

```
// ThisInstrLength()
// =====
// Return the length of the current instruction.

impdef func ThisInstrLength() => integer
begin
    Unimplemented();
end;
```

Library pseudocode for shared/functions/system/UndefinedInjectionCheck

```
// UndefinedInjectionCheck()
// =====
// Check PSTATE.UINJ to determine if execution of the current
// instruction should cause an Undefined exception.

func UndefinedInjectionCheck()
begin
    if IsFeatureImplemented(FEAT_UINJ) && PSTATE.UINJ == '1' then
        Undefined();
    end;
end;
```

Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()
// =====
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64_

readonly func UsingAArch32() => boolean
begin
    let aarch32 : boolean = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32; end;
    if !HaveAArch64() then assert aarch32; end;
    return aarch32;
end;
```


Library pseudocode for shared/functions/system/ValidSecurityStateAtEL

```
// ValidSecurityStateAtEL()
// =====
// Returns TRUE if the current settings and architecture choices for this
// implementation permit a valid Security state at the indicated EL.

func ValidSecurityStateAtEL(el : bits(2)) => boolean
begin
    if !HaveEL(el) then
        return FALSE;
    end;

    if el == EL3 then
        return TRUE;
    end;

    if IsFeatureImplemented(FEAT_RME) then
        let effective_nse_ns : bits(2) = EffectiveSCR_EL3_NSE() :: EffectiveSCR_EL3_NS();
        if effective_nse_ns == '10' then
            return FALSE;
        end;
    end;

    if el == EL2 then
        return EL2Enabled();
    end;

    return TRUE;
end;
```

Library pseudocode for shared/functions/system/VirtualFIQPending

```
// VirtualFIQPending()
// =====
// Returns TRUE if there is any pending virtual FIQ.

impdef func VirtualFIQPending() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/system/VirtualIRQPending

```
// VirtualIRQPending()
// =====
// Returns TRUE if there is any pending virtual IRQ.

impdef func VirtualIRQPending() => boolean
begin
    return FALSE;
end;
```

Library pseudocode for shared/functions/system/WFxType

```
// WFxType
// =====
// WFx instruction types.

type WFxType of enumeration {WfxType_WFE, WfxType_WFI, WfxType_WFET, WfxType_WFIT};
```

Library pseudocode for shared/functions/system/WaitForEvent

```
// WaitForEvent()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

func WaitForEvent()
begin
    if Halted() then return; end;
    if !IsEventRegisterSet() then
        EnterLowPowerState();
    end;
    return;
end;

// WaitForEvent()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

func WaitForEvent(localtimeout : integer)
begin
    if Halted() then return; end;
    if !\(IsEventRegisterSet\) || LocalTimeoutEvent\(localtimeout\) then
        EnterLowPowerState();
    end;
    return;
end;
```

Library pseudocode for shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.

func WaitForInterrupt()
begin
    if Halted() then return; end;
    EnterLowPowerState();
    return;
end;

// WaitForInterrupt()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.

func WaitForInterrupt(localtimeout : integer)
begin
    if Halted() then return; end;
    if !LocalTimeoutEvent(localtimeout) then
        EnterLowPowerState();
    end;
    return;
end;
```

Library pseudocode for shared/functions/system/WatchpointRelatedSyndrome

```
// WatchpointRelatedSyndrome()
// =====
// Update common Watchpoint related fields.

func WatchpointRelatedSyndrome(fault : FaultRecord) => bits(24)
begin
    var syndrome : bits(24) = Zeros{};

    if fault.watchptinfo.maybe_false_match then
        syndrome[16] = '1'; // WPF
    elsif IsFeatureImplemented(FEAT_Debugv8p2) then
        syndrome[16] = ImpDefBit("WPF value on TRUE Watchpoint match");
    end;

    if IsRelaxedWatchpointAccess(fault.accessdesc) then
        if HaltOnBreakpointOrWatchpoint() then
            if ImpDefBool("EDWAR is not valid on watchpoint debug event") then
                syndrome[10] = '1'; // FnV
            end;
        else
            if ImpDefBool("FAR is not valid on watchpoint exception") then
                syndrome[10] = '1'; // FnV
            end;
        end;
    else
        if WatchpointFARNotPrecise(fault) then
            syndrome[15] = '1'; // FnP
        end;
    end;

    // Watchpoint number is valid if FEAT_Debugv8p9 is implemented or
    // if Feat_Debugv8p2 is implemented and below set of conditions are satisfied:
    // - Either FnV = 1 or FnP = 1.
    // - If the address recorded in FAR is not within a naturally-aligned block of memory.
    // Otherwise, it is IMPLEMENTATION DEFINED if watchpoint number is valid.
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        syndrome[17] = '1'; // WPTV
        syndrome[23:18] = fault.watchptinfo.watchpt_num[5:0]; // WPT
    elsif IsFeatureImplemented(FEAT_Debugv8p2) then
        if syndrome[15] == '1' || syndrome[10] == '1' then // Either of FnP or FnV is 1
            syndrome[17] = '1'; // WPTV
        elsif !AddressInNaturallyAlignedBlock(fault.vaddress, fault.watchptinfo.vaddress) then
            syndrome[17] = '1'; // WPTV
        elsif ImpDefBool("WPTV field is valid") then
            syndrome[17] = '1';
        end;
        if syndrome[17] == '1' then
            syndrome[23:18] = fault.watchptinfo.watchpt_num[5:0]; // WPT
        else
            syndrome[23:18] = ARBITRARY : bits(6);
        end;
    end;

    return syndrome;
end;
```

Library pseudocode for shared/functions/tlbi

```
constant VMID_NONE : bits(NUM_VMIDBITS) = Zeros{};
constant ASID_NONE : bits(NUM_ASIDBITS) = Zeros{};
```

Library pseudocode for shared/functions/tlbi/Broadcast

```
// Broadcast
// =====

type Broadcast of enumeration {
    Broadcast_ISH,
    Broadcast_ForcedISH,
    Broadcast_OSH,
    Broadcast_NSH
};
```

Library pseudocode for shared/functions/tlbi/BroadcastTLBI

```
// BroadcastTLBI()
// =====
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated broadcast
// domain.

impdef func BroadcastTLBI(broadcast : Broadcast, r : TLBIRecord)
begin
    return;
end;
```

Library pseudocode for shared/functions/tlbi/ExcludeXS

```
// ExcludeXS()
// =====
// Returns TRUE if a TLBI maintenance instruction without the nXS qualifier executed at EL1
// behaves in the same way as the corresponding TLBI maintenance instruction with the nXS qualifier.

func ExcludeXS() => boolean
begin
    return (IsFeatureImplemented(FEAT_XS) && PSTATE.EL == EL1 &&
            IsHCRXEL2Enabled() && HCRX_EL2().FnXS == '1');
end;
```

Library pseudocode for shared/functions/tlbi/TLBI

```
// TLBI()
// =====
// Invalidates TLB entries for which TLBIMatch() returns TRUE.

impdef func TLBI(r : TLBIRecord)
begin
    return;
end;
```

Library pseudocode for shared/functions/tlbi/TLBILevel

```
// TLBILevel
// =====

type TLBILevel of enumeration {
    TLBILevel_Any,           // this applies to TLB entries at all levels
    TLBILevel_Last          // this applies to TLB entries at last level only
};
```

Library pseudocode for shared/functions/tlbi/TLBIMemAttr

```
// TLBIMemAttr
// =====
// Defines the attributes of the memory operations that must be completed in
// order to deem the TLBI operation as completed.

type TLBIMemAttr of enumeration {
    TLBI_AllAttr,           // All TLB entries within the scope of the invalidation
    TLBI_ExcludeXS         // Only TLB entries with XS=0 within the scope of the invalidation
};
```

Library pseudocode for shared/functions/tlbi/TLBIOp

```
// TLBIOp
// =====

type TLBIOp of enumeration {
    TLBIOp_DALL,           // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,           // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIOp_RPA,
    TLBIOp_PAALL,
    TLBIOp_VMALLWS2
};
```

Library pseudocode for shared/functions/tlbi/TLBIRecord

```
// TLBIRecord
// =====
// Details related to a TLBI operation.

type TLBIRecord of record {
    op : TLBIOp,
    from_aarch64 : boolean, // originated as an AArch64 operation
    security : SecurityState,
    regime : Regime,
    use_vmid : boolean,
    vmid : bits(NUM_VMIDBITS),
    asid : bits(NUM_ASIDBITS),
    level : TLBILevel,
    attr : TLBIMemAttr,
    ipaspace : PASpace, // For operations that take IPA as input address
    address : bits(64), // input address, for range operations, start address
    end_address : bits(64), // for range operations, end; address
    d64 : boolean, // For operations that evict VMSAv8-64 based TLB entries
    d128 : boolean, // For operations that evict VMSAv9-128 based TLB entries
    ttl : bits(4), // translation table walk level holding the leaf entry
                    // for the address being invalidated
                    // For Non-Range Invalidations:
                    //   When the ttl is
                    //   '00xx' : this applies to all TLB entries
                    //   Otherwise : TLBIP instructions invalidates D128 TLB
                    //   entries only
                    //   TLBI instructions invalidates D64 TLB
                    //   entries only
                    // For Range Invalidations:
                    //   When the ttl is
                    //   '00' : this applies to all TLB entries
                    //   Otherwise : TLBIP instructions invalidates D128 TLB
                    //   entries only
                    //   TLBI instructions invalidates D64 TLB
                    //   entries only
    tg : bits(2) // for range operations, translation granule
};
```

Library pseudocode for shared/functions/tlbi/VMID

```
// VMID
// ====
// Effective VMID.

func VMID() => bits(NUM_VMIDBITS)
begin
  if EL2Enabled\(\) then
    if !ELUsingAArch32\(EL2\) then
      if IsFeatureImplemented(FEAT_VMID16) && VTCR_EL2().VS == '1' then
        return VTTBR_EL2().VMID;
      else
        return ZeroExtend{NUM_VMIDBITS}(VTTBR_EL2().VMID[7:0]);
      end;
    else
      return ZeroExtend{NUM_VMIDBITS}(VTTBR().VMID);
    end;
  elseif HaveEL\(EL2\) && IsFeatureImplemented(FEAT_SEL2) then
    return Zeros{NUM_VMIDBITS};
  else
    return VMID\_NONE;
  end;
end;
```

Library pseudocode for shared/functions/unbounded_configs/Maximum

```
// Maximum walk levels
// =====
let MAX_WALK_LEVELS : integer{} = (if IsFeatureImplemented(FEAT_D128) then 6
                                     else (if IsFeatureImplemented(FEAT_LPA2) then 5
                                             else 4));
```

Library pseudocode for shared/functions/unbounded_configs/Unbounded

```
// Unbounded Variables
// =====
// Sets the upper limit for a specific loop or function.

constant Unbounded : integer{} = 2^128;
config Unbounded_DiscardTransactionalWrites : integer{0..Unbounded} = Unbounded;
config Unbounded_DescriptorUpdate : integer{0..Unbounded} = Unbounded;
config Unbounded_semihost : integer{0..Unbounded} = Unbounded;
config Unbounded_ParseTrace : integer{0..Unbounded} = Unbounded;
config Unbounded_ParseExtensionPacket : integer{0..Unbounded} = Unbounded;
config Unbounded_ULEB128 : integer{0..Unbounded} = Unbounded;
config Unbounded_ULEB128n : integer{0..Unbounded} = Unbounded;
config Unbounded_FloorPow2 : integer{0..Unbounded} = Unbounded;
config Unbounded_NormalizeReal : integer{0..Unbounded} = Unbounded;
```



```

// ConstrainUnpredictable()
// =====
// Return the appropriate Constraint result to control the caller's behavior.
// The return value is IMPLEMENTATION DEFINED within a permitted list for each
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

readonly impdef func ConstrainUnpredictable(which : Unpredictable) => Constraint
begin
  case which of
    when Unpredictable VMSR =>
      return Constraint UNDEF;
    when Unpredictable WBOVERLAPLD =>
      return Constraint WBSUPPRESS; // return loaded value
    when Unpredictable WBOVERLAPST =>
      return Constraint NONE; // store pre-writeback value
    when Unpredictable LDPOVERLAP =>
      return Constraint UNDEF; // instruction is UNDEFINED
    when Unpredictable BASEOVERLAP =>
      return Constraint UNKNOWN; // use UNKNOWN address
    when Unpredictable DATAOVERLAP =>
      return Constraint UNKNOWN; // store UNKNOWN value
    when Unpredictable DEVPAGE2 =>
      return Constraint FAULT; // take an alignment fault
    when Unpredictable RESTCF =>
      return Constraint UNKNOWN; // Do not take a fault
    when Unpredictable DEVICETAGSTORE =>
      return Constraint NONE; // Do not take a fault
    when Unpredictable INSTRDEVICE =>
      return Constraint NONE; // Do not take a fault
    when Unpredictable RESCPACR =>
      return Constraint TRUE; // Map to UNKNOWN value
    when Unpredictable RESMAIR =>
      return Constraint UNKNOWN; // Map to UNKNOWN value
    when Unpredictable SICTAGGED =>
      return Constraint FALSE; // SCTLR_ELx.C == '0' marks address as untagged
    when Unpredictable S2RESMEMATTR =>
      return Constraint NC; // Map to Noncacheable value
    when Unpredictable RESTEXCB =>
      return Constraint UNKNOWN; // Map to UNKNOWN value
    when Unpredictable RESDACR =>
      return Constraint UNKNOWN; // Map to UNKNOWN value
    when Unpredictable RESPRRR =>
      return Constraint UNKNOWN; // Map to UNKNOWN value
    when Unpredictable RESVTCRS =>
      return Constraint UNKNOWN; // Map to UNKNOWN value
    when Unpredictable RESTnSZ =>
      return Constraint FORCE; // Map to the limit value
    when Unpredictable OORTnSZ =>
      return Constraint FORCE; // Map to the limit value
    when Unpredictable LARGEIPA =>
      return Constraint FORCE; // Restrict the IA size to the PAMax value
    when Unpredictable ESRCONDPASS =>
      return Constraint FALSE; // Report as "AL"
    when Unpredictable ILZEROIT =>
      return Constraint FALSE; // Do not zero PSTATE.IT
    when Unpredictable ILZEROT =>
      return Constraint FALSE; // Do not zero PSTATE.T
    when Unpredictable BPVECTORCATCHPRI =>
      return Constraint TRUE; // Debug Vector Catch: match on 2nd halfword
    when Unpredictable VCMATCHHALF =>
      return Constraint FALSE; // No match
    when Unpredictable VCMATCHDAPA =>
      return Constraint FALSE; // No match on Data Abort or Prefetch abort
    when Unpredictable WPMASKANDBAS =>
      return Constraint FALSE; // Watchpoint disabled
    when Unpredictable WPBASCONTIGUOUS =>
      return Constraint FALSE; // Watchpoint disabled
    when Unpredictable RESWPMASK =>
      return Constraint DISABLED; // Watchpoint disabled
    when Unpredictable WPMASKEDBITS =>
      return Constraint FALSE; // Watchpoint disabled
    when Unpredictable RESBPWPCTRL =>
      return Constraint DISABLED; // Breakpoint/watchpoint disabled
    when Unpredictable BPNOTIMPL =>
      return Constraint DISABLED; // Breakpoint disabled
    when Unpredictable RESBPTYPE =>
      return Constraint DISABLED; // Breakpoint disabled
    when Unpredictable BPNOTCTXCMP =>
      return Constraint DISABLED; // Breakpoint disabled
    when Unpredictable RESMDSELR =>
      return Constraint UNKNOWN; // Map to UNKNOWN value
    when Unpredictable BPMATCHHALF =>
      return Constraint FALSE; // No match
    when Unpredictable BPMISMATCHHALF =>
      return Constraint FALSE; // No match
    when Unpredictable BPLINKINGDISABLED =>
      return Constraint FALSE; // Breakpoint does not match
    when Unpredictable RESTARTALIGNPC =>

```



```

    return Constraint FALSE; // Do not force alignment
when Unpredictable RESTARTZEROUPPERPC =>
    return Constraint TRUE; // Force zero extension
when Unpredictable ZEROUPPER =>
    return Constraint TRUE; // zero top halves of X registers
when Unpredictable ERETZEROUPPERPC =>
    return Constraint TRUE; // zero top half of PC
when Unpredictable A32FORCEALIGNPC =>
    return Constraint FALSE; // Do not force alignment
when Unpredictable SMD =>
    return Constraint UNDEF; // disabled SMC is Unallocated
when Unpredictable NONFAULT =>
    return Constraint FALSE; // Speculation enabled
when Unpredictable SVEZEROUPPER =>
    return Constraint TRUE; // zero top bits of Z registers
when Unpredictable SVELDNFDATA =>
    return Constraint TRUE; // Load mem data in NF loads
when Unpredictable SVELDNFZERO =>
    return Constraint TRUE; // Write zeros in NF loads
when Unpredictable CHECKSPNONEACTIVE =>
    return Constraint TRUE; // Check SP alignment
when Unpredictable SMEZEROUPPER =>
    return Constraint TRUE; // zero top bits of ZA registers
when Unpredictable NVNV1 =>
    return Constraint NVNV1_00; // Map unpredictable configuration of HCR_EL2[NV,NV1]
                                // to NV = 0 and NV1 = 0
when Unpredictable BADDR_RESS =>
    return Constraint RESS; // Values behave as RESS
when Unpredictable Shareability =>
    return Constraint OSH; // Map reserved encoding of shareability to Outer Shareable
when Unpredictable AFUPDATE => // AF update for alignment or Permission fault
    return Constraint TRUE;
when Unpredictable DBUPDATE => // DB State update for alignment fault
    return Constraint TRUE;
when Unpredictable Unsupported Atomic HW Update =>
    return Constraint FALSE; // Do not raise Unsupported Atomic HW Update fault
                                // when encountering unsupported memory attributes
when Unpredictable IESBinDebug => // Use SCTLx_ELx[].IESB in Debug state
    return Constraint TRUE;
when Unpredictable BADPMSFCR => // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
    return Constraint TRUE;
when Unpredictable RES_PMU_VS => // Bad setting for SVE Streaming mode filter
    return Constraint FALSE;
when Unpredictable COUNT_CHAIN => // Chained PMU counters idx, idx+1 are not in same range
    return Constraint FALSE;
when Unpredictable EL1TIMESTAMP => // Bad settings for TRFCR_EL1
    return Constraint EL1TIMESTAMP;
when Unpredictable EL2TIMESTAMP => // Bad settings for TRFCR_EL2
    return Constraint EL2TIMESTAMP;
when Unpredictable ZEROBTYP =>
    return Constraint TRUE; // Save BTYPE in SPSR_ELx/DPSR_EL0 as '00'
when Unpredictable CLEARERRITEZERO => // Clearing sticky errors when instruction in flight
    return Constraint FALSE;
when Unpredictable ALUEXCEPTIONRETURN =>
    return Constraint UNDEF;
when Unpredictable DBGxVR_RESS =>
    return Constraint FALSE;
when Unpredictable PMSCR_PCT =>
    return Constraint PMSCR_PCT_VIRT;
when Unpredictable WFXTDEBUG =>
    return Constraint FALSE; // WFXT in Debug state does not execute as a NOP
// Accesses are not single-copy atomic above the byte level
when Unpredictable LS64UNSUPPORTED =>
    return Constraint LIMITED ATOMICITY;
// Unaligned exclusives, atomics, acquire/release
// to a region that is not to Normal inner write-back
// outer write-back shareable generate an Alignment fault.
when Unpredictable LSE2_ALIGNMENT_FAULT =>
    return Constraint FALSE;
when Unpredictable LSE128OVERLAP =>
    return Constraint UNDEF; // instruction is UNDEFINED
when Unpredictable IGNORETRAPINDEBUG =>
    return Constraint FALSE; // Trap to register access in debug state is enabled
when Unpredictable PMUEVENTCOUNTER =>
    return Constraint UNDEF; // Accesses to the register are UNDEFINED
when Unpredictable RES_HPMN =>
    return Constraint UNKNOWN; // The counter is reserved for EL2 use
when Unpredictable RES_EPMN =>
    return Constraint UNKNOWN; // The counter is reserved for external use
when Unpredictable BRBFILTRATE =>
    return Constraint FALSE; // BRB_FILTRATE event not generated on BRB injection
when Unpredictable PMUSNAPSHOTEVENT => // PMU_SNAPSHOT event not generated in Debug state
    return Constraint FALSE;
when Unpredictable RESEPMSSAD =>
    // External access to PMUv3 Snapshot extension allowed in Secure state
    // when FEAT_RME is not implemented or Root state otherwise
    return Constraint UNKNOWN;
when Unpredictable RESPMSSSE =>
    return Constraint DISABLED; // PMU capture events disabled

```

```

when Unpredictable RESPMEE =>
    return Constraint DISABLED; // PMU Profiling exception disabled, PMUIRQ enabled
when Unpredictable RESPMSEE =>
    return Constraint DISABLED; // SPE Profiling exception disabled
when Unpredictable RESTRFEE =>
    return Constraint DISABLED; // TRBE Profiling exception disabled
when Unpredictable RESTC =>
    return Constraint DISABLED; // Threshold features disabled
when Unpredictable MOPSOVERLAP => // Instruction is UNDEFINED
    return Constraint UNDEF;
when Unpredictable MOPS R31 => // Instruction is UNDEFINED
    return Constraint UNDEF;
when Unpredictable CASRETURNOLDVALUE =>
    return Constraint TRUE;
when Unpredictable WRITEFAILED CAS =>
    return Constraint FALSE;
when Unpredictable WRITEFAILED RCW CAS =>
    return Constraint FALSE;
when Unpredictable STRONLYTAGCHECKED CAS =>
    return Constraint TRUE; // CAS with compare fail does a Tag Check
when Unpredictable STRONLYTAGCHECKED RCW SCAS =>
    return Constraint TRUE; // RCW(S)CAS with RCW(S) fail/compare fail does a Tag Check
when Unpredictable RESERVEDNSxB =>
    return Constraint MAPTOALLOTTED;
when Unpredictable RESERVEDNSxB Trap =>
    return Constraint TRUE;
when Unpredictable RES ETBAD =>
    // ExternalTracebufferaccess disabled for res values
    return Constraint DISABLED;
when Unpredictable RESBPMASK =>
    return Constraint DISABLED; // Mask set to 0
when Unpredictable BPMASK =>
    return Constraint FALSE; // Breakpoint match will fail
when Unpredictable BPMASKEDBITS =>
    return Constraint FALSE; // Breakpoint match will fail
when Unpredictable BPLINKEDADDRMATCH =>
    return Constraint FALSE; // Breakpoint match will fail
when Unpredictable 16BYTEROUNDEDUPACCESS =>
    // Watchpoint match of 16 byte rounded range in case of SVE access
    return Constraint TRUE;
when Unpredictable 16BYTEROUNDEDDOWNACCESS =>
    // Watchpoint match of 16 byte rounded range in case of SVE access
    return Constraint TRUE;
when Unpredictable NODTRTAGCHK =>
    // The load or store operation does not perform the Tag check in Debug state
    return Constraint TRUE;
when Unpredictable CPACHECK => // Do not perform Checked Pointer Arithmetic
    return Constraint FALSE;
when Unpredictable Atomic SYNC ABORT =>
    return Constraint FALSE;
when Unpredictable Atomic ERROR =>
    return Constraint FALSE;
when Unpredictable Atomic NOP =>
    return Constraint FALSE;
when Unpredictable Atomic MMU IMPDEF FAULT =>
    return Constraint FALSE;
when Unpredictable USE DEFAULT PMG =>
    return Constraint TRUE;
when Unpredictable BankedRegister =>
    return Constraint NOP;
when Unpredictable UnimplementedRegister =>
    return Constraint NOP;
when Unpredictable RES RA =>
    return Constraint TRUE;
when Unpredictable BPNOTLINKING =>
    return Constraint FALSE;
otherwise =>
    unreachable;
end;
end;

```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

readonly impdef func ConstrainUnpredictableBits(width)(which : Unpredictable
                                                    ) => (Constraint, bits(width))

begin
  let c : Constraint = ConstrainUnpredictable(which);

  case c of
    when Constraint_UNKNOWN =>
      return (c, Zeros{width}); // See notes; this is an example implementation only
    when Constraint_PMSCR_PCT_VIRT =>
      return (c, Zeros{width});
    when Constraint_NC where which == Unpredictable_S2RESMEMATTR =>
      assert width == 4;
      return (c, ZeroExtend{width}('0101'));
    when Constraint_WT where which == Unpredictable_S2RESMEMATTR =>
      assert width == 4;
      return (c, ZeroExtend{width}('1010'));
    when Constraint_WB where which == Unpredictable_S2RESMEMATTR =>
      assert width == 4;
      return (c, ZeroExtend{width}('1111'));
    when Constraint_EL1TIMESTAMP =>
      return (c, ZeroExtend{width}('01'));
    when Constraint_EL2TIMESTAMP =>
      return (c, ZeroExtend{width}('01'));
    when Constraint_MAPTOALLOCATED =>
      return (c, ZeroExtend{width}('010')); // return non-secure state_bits
    otherwise =>
      return (c, ARBITRARY : bits(width)); // bits result not used

  end;
end;
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// =====
// This is a variant of the ConstrainUnpredictable function where the result is either
// Constraint_TRUE or Constraint_FALSE.

impdef func ConstrainUnpredictableBool(which : Unpredictable) => boolean

begin
  let c : Constraint = ConstrainUnpredictable(which);
  assert c IN {Constraint_TRUE, Constraint_FALSE};
  return (c == Constraint_TRUE);
end;
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// =====
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN
// value in the range low to high, inclusive.

impdef func ConstrainUnpredictableInteger(low : integer,
                                           high : integer,
                                           which : Unpredictable) => (Constraint, integer)

begin
  let c : Constraint = ConstrainUnpredictable(which);
  if c == Constraint_UNKNOWN then
    return (c, low); // See notes; this is an example implementation only
  else
    return (c, ARBITRARY : integer); // integer result not used
  end;
end;
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableProcedure

```
// ConstrainUnpredictableProcedure()
// =====
// This is a variant of ConstrainUnpredictable that implements a Constrained
// Unpredictable behavior for a given Unpredictable situation.
// The behavior is within permitted behaviors for a given Unpredictable situation,
// these are documented in the textual part of the architecture specification.
//
// This function is expected to be refined in an IMPLEMENTATION DEFINED manner.
// The details of possible outcomes may not be present in the code and must be interpreted
// for each use with respect to the CONSTRAINED UNPREDICTABLE specifications
// for the specific area.

impdef func ConstrainUnpredictableProcedure(which : Unpredictable)
begin
    case which of
        when Unpredictable PMUEVENTCOUNTER =>
            Undefined();
        when Unpredictable MRC APSR TARGET =>
            PSTATE.[N,Z,C,V] = DBGDSRint() [31:28];
        otherwise =>
            ImpDef("IMPLEMENTATION_DEFINED");
    end;
    return;
end;
```

Library pseudocode for shared/functions/unpredictable/Constraint

```
// Constraint
// =====
// List of Constrained Unpredictable behaviors.

type Constraint of enumeration {
    // General
    Constraint_NONE, // Instruction executes with
                    // no change or side-effect
                    // to its described behavior
    Constraint_UNKNOWN, // Destination register
                    // has UNKNOWN value
    Constraint_UNDEF, // Instruction is UNDEFINED
    Constraint_UNDEFELO, // Instruction is UNDEFINED at EL0 only
    Constraint_NOP, // Instruction executes as NOP
    Constraint_TRUE,
    Constraint_FALSE,
    Constraint_DISABLED,
    Constraint_UNCOND, // Instruction executes unconditionally
    Constraint_COND, // Instruction executes conditionally
    Constraint_ADDITIONAL_DECODE, // Instruction executes
                                // with additional decode
    // Load-store
    Constraint_WBSUPPRESS,
    Constraint_FAULT,
    Constraint_LIMITED_ATOMICITY, // Accesses are not
                                // single-copy atomic
                                // above the byte level
    Constraint_NVNV1_00,
    Constraint_NVNV1_01,
    Constraint_NVNV1_11,
    Constraint_EL1TIMESTAMP, // Constrain to Virtual Timestamp
    Constraint_EL2TIMESTAMP, // Constrain to Virtual Timestamp
    Constraint_OSH, // Constrain to Outer Shareable
    Constraint_ISH, // Constrain to Inner Shareable
    Constraint_NSH, // Constrain to Nonshareable
    Constraint_NC, // Constrain to Noncacheable
    Constraint_WT, // Constrain to Writethrough
    Constraint_WB, // Constrain to Writeback
    Constraint_RESS, // Bits behave as RESS for all purposes
                    // other than reading back the register
    Constraint_ALLRESS, // Bits behave as RESS for all purposes
    // IPA too large
    Constraint_FORCE, Constraint_FORCENOSLCHECK,
    // An unallocated System register value maps onto an allocated value
    Constraint_MAPTOALLOCATED,
    // PMSCR_PCT reserved values select Virtual timestamp
    Constraint_PMSCR_PCT_VIRT,
    Constraint_ANYREG, // Any allocated register is chosen
};
```



```

// Unpredictable
// =====
// List of Constrained Unpredictable situations.

type Unpredictable of enumeration {
    // VMSR on MVFR
    Unpredictable_VMSR,
    // Writeback/transfer register overlap (load)
    Unpredictable_WBOVERLAPLD,
    // Writeback/transfer register overlap (store)
    Unpredictable_WBOVERLAPST,
    // Load Pair transfer register overlap
    Unpredictable_LDPOVERLAP,
    // Store-exclusive base/status register overlap
    Unpredictable_BASEOVERLAP,
    // Store-exclusive data/status register overlap
    Unpredictable_DATAOVERLAP,
    // Load-store alignment checks
    Unpredictable_DEVPAGE2,
    // Instruction fetch from Device memory
    Unpredictable_INSTRDEVICE,
    // Reserved CPACR value
    Unpredictable_RESCPACR,
    // Reserved MAIR value
    Unpredictable_RESMAIR,
    // Effect of SCTLr_ELx.C on Tagged attribute
    Unpredictable_S1CTAGGED,
    // Reserved Stage 2 MemAttr value
    Unpredictable_S2RESMEMATTR,
    // Reserved TEX::C::B value
    Unpredictable_RESTEXCB,
    // Reserved PRRR value
    Unpredictable_RESRRR,
    // Reserved DACR field
    Unpredictable_RESDACR,
    // Reserved VTCR.S value
    Unpredictable_RESVTCRS,
    // Reserved TCR.TnSZ value
    Unpredictable_RESTnSZ,
    // Reserved SCTLr_ELx.TCF value
    Unpredictable_RESTCF,
    // Tag stored to Device memory
    Unpredictable_DEVICETAGSTORE,
    // Out-of-range TCR.TnSZ value
    Unpredictable_OORTnSZ,

    // IPA size exceeds PA size
    Unpredictable_LARGEIPA,
    // Syndrome for a known-passing conditional A32 instruction
    Unpredictable_ESRCONDPASS,
    // Illegal State exception: zero PSTATE.IT
    Unpredictable_ILZEROIT,
    // Illegal State exception: zero PSTATE.T
    Unpredictable_ILZEROT,
    // Debug: prioritization of Vector Catch
    Unpredictable_BPVECTORCATCHPRI,
    // Debug Vector Catch: match on 2nd halfword
    Unpredictable_VCMATCHHALF,
    // Debug Vector Catch: match on Data Abort
    // or Prefetch abort
    Unpredictable_VCMATCHDAPA,
    // Debug watchpoints: nonzero MASK and non-ones BAS
    Unpredictable_WPMASKANDBAS,
    // Debug watchpoints: non-contiguous BAS
    Unpredictable_WPBASCONTIGUOUS,
    // Debug watchpoints: reserved MASK
    Unpredictable_RESWPMASK,
    // Debug watchpoints: nonzero MASKed bits of address
    Unpredictable_WPMASKEDBITS,
    // Debug breakpoints and watchpoints: reserved control bits
    Unpredictable_RESBPWPCTRL,
    // Debug breakpoints: not implemented
    Unpredictable_BPNOTIMPL,
    // Debug breakpoints: reserved type
    Unpredictable_RESBPTYPE,
    // Debug breakpoints and watchpoints: reserved MDSELR_EL1.BANK
    Unpredictable_RESMDSELR,
    // Debug breakpoints: not-context-aware breakpoint
    Unpredictable_BPNOTCTXCMP,
    // Debug breakpoints: match on 2nd halfword of instruction
    Unpredictable_BPMATCHHALF,
    // Debug breakpoints: mismatch on 2nd halfword of instruction
    Unpredictable_BPMISMATCHHALF,
    // Debug breakpoints: a breakpoint is linked to that is not
    // programmed with linking enabled
    Unpredictable_BPLINKINGDISABLED,
    // Debug breakpoints: reserved MASK
    Unpredictable_RESBPMASK,
    // Debug breakpoints: MASK is set for a Context matching

```

```

// breakpoint or when DBGBCR_EL1(n).BAS != '1111'
Unpredictable_BPMASK,
// Debug breakpoints: nonzero MASKed bits of address
Unpredictable_BPMASKEDBITS,
// Debug breakpoints: A linked breakpoint is
// linked to an address matching breakpoint
Unpredictable_BPLINKEDADDRMATCH,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
// Zero top 32 bits of PC on illegal return to
// AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking
// branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// FF speculation
Unpredictable_NONFAULT,
// Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
// Load mem data in NF loads
Unpredictable_SVELDNFDATA,
// Write zeros in NF loads
Unpredictable_SVELDNFZERO,
// SP alignment fault when predicate is all zero
Unpredictable_CHECKSPNONEACTIVE,
// Zero top bits of ZA registers in EL change
Unpredictable_SMEZEROUPPER,
// Watchpoint match of last rounded up memory access in case of
// 16 byte rounding
Unpredictable_16BYTEROUNDEDUPACCESS,
// Watchpoint match of first rounded down memory access in case of
// 16 byte rounding
Unpredictable_16BYTEROUNDEDDOWNACCESS,
// HCR_EL2.[NV,NV1] == '01'
Unpredictable_NVNV1,
// Upper bits of a BADDR are not RESS
Unpredictable_BADDR_RESS,
// Reserved shareability encoding
Unpredictable_Shareability,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Dirty Bit State Update by HW
Unpredictable_DBUPDATE,
// Unsupported memory attributes for HW update
Unpredictable_Unsupported_Atomic_HW_Update,
// Consider SCTLX_ELx().IESB in Debug state
Unpredictable_IESBinDebug,
// The HCR_EL2.TERR / HCR2.TERR and SCR_EL3.TERR / SCR.TERR
// trap control can be treated as RES0.
Unpredictable_MINIMALRASv2,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYP,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
// Reserved MDCR_EL3.[NSTBE,NSTB] or MDCR_EL3.[NSPBE,NSPB] value
Unpredictable_RESERVEDNSxB,
Unpredictable_RESERVEDNSxB_Trap,
// WFET or WFIT instruction in Debug state
Unpredictable_WFXTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Unaligned exclusives, atomics, acquire/release
// to a region that is not to Normal inner write-back
// outer write-back shareable generate an Alignment fault.
Unpredictable_LSE2_ALIGNMENT_FAULT,
// 128-bit Atomic or 128-bit RCW{S} transfer register overlap
Unpredictable_LSE128OVERLAP,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
// ALUEXCEPTIONRETURN when in user/system mode in
// A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
// Compare DBGVCR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,
// Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,
// Reserved PMSCR.PCT behavior
Unpredictable_PMSCR_PCT,

```

```

// MDCR_EL2.HPMN or HDCR.HPMN is larger than PMCR.N or
// FEAT_HPMN0 is not implemented and HPMN is 0.
Unpredictable_RES_HPMN,
// Chained PMU counters idx and idx+1 are not in same range
Unpredictable_COUNT_CHAIN,
// PMCCR.EPMN is larger than PMCR.N
Unpredictable_RES_EPMN,
// Generate BRB_FILTRATE event on BRB injection
Unpredictable_BRBFILTRATE,
// Generate PMU_SNAPSHOT event in Debug state
Unpredictable_PMUSNAPSHOTEVENT,
// Reserved MDCR_EL3.EPMSSAD value
Unpredictable_RESEPMSSAD,
// Reserved PMECR_EL1.SSE value
Unpredictable_RESPMSSE,
// Enable for PMU Profiling exception and PMUIRQ
Unpredictable_RESPMEE,
// Enables for SPE Profiling exceptions and PMSIRQ
Unpredictable_RESPMSEE,
// Enables for TRBE Profiling exceptions and PMSIRQ
Unpredictable_RESTRFEE,
// Operands for CPY*/SET* instructions overlap
Unpredictable_MOPSOVERLAP,
// Operands for CPY*/SET* instructions use 0b11111
// as a register specifier
Unpredictable_MOPS_R31,
// Chooses which value to return in a non failed Atomic Compare and Swap
Unpredictable_CASRETURNOLDVALUE,
// Enables write of the newvalue in a failed Atomic Compare and Swap
Unpredictable_WRITEFAILEDCAS,
// Enables write of the newvalue in a failed Atomic Compare and Swap
// or failed RCW{S}
Unpredictable_WRITEFAILEDRCWCAS,
// Store-only Tag checking on a failed Atomic Compare and Swap
Unpredictable_STRONLYTAGCHECKEDCAS,
// Store-only Tag checking on a failed RCW(S) checks
// or RCW(S) Atomic Compare and Swap
Unpredictable_STRONLYTAGCHECKEDRCWSCAS,
// Reserved MDCR_EL3.ETBAD value
Unpredictable_RES_ETBAD,
// Invalid Streaming Mode filter bits
Unpredictable_RES_PMU_VS,
// Apply Checked Pointer Arithmetic on a sequential access to bytes
// that cross the 0xFFFF_FFFF_FFFF_FFFF boundary.
Unpredictable_CPACHECK,
// Reserved PMEVTYPER[n]_EL0.[TC,TE,TC2] values
Unpredictable_RESTC,
// When FEAT_MTE is implemented, if Memory access mode is enabled
// and PSTATE.TCO is 0, Reads and writes to the external debug
// interface DTR registers are CONSTRAINED UNPREDICTABLE for tagcheck
Unpredictable_NODTRTAGCHK,
// Use the default PMG when the default PARTID is generated
// due to MPAM error
Unpredictable_USE_DEFAULT_PMG,
// If the atomic instructions are not atomic in regard to other
// agents that access memory, then the instruction can have one or
// more of the following effects
Unpredictable_Atomic_SYNC_ABORT,
Unpredictable_Atomic_ERROR,
Unpredictable_Atomic_NOP,
Unpredictable_Atomic_MMU_IMPDEF_FAULT,
// Accessing DBGDSCRint via MRC in debug state
Unpredictable_MRC_APSR_TARGET,
// Accessing Banked register not accessible from the PE mode
Unpredictable_BankedRegister,
// Accessing unimplemented Banked register
Unpredictable_UnimplementedRegister,
// Reserved ERRACR.{RLRA,SRA,NSRA} values
Unpredictable_RES_RA,
// Linked target does not support linking
Unpredictable_BPNOTLINKING
};

```


Library pseudocode for shared/functions/vector/AdvSIMDEExpandImm

```
// AdvSIMDEExpandImm()
// =====

func AdvSIMDEExpandImm(op : bit, cmode : bits(4), imm8 : bits(8)) => bits(64)
begin
    var imm64 : bits(64);
    case cmode[3:1] of
        when '000' =>
            imm64 = Replicate{64}(Zeros{24}::imm8);
        when '001' =>
            imm64 = Replicate{64}(Zeros{16}::imm8::Zeros{8});
        when '010' =>
            imm64 = Replicate{64}(Zeros{8}::imm8::Zeros{16});
        when '011' =>
            imm64 = Replicate{64}(imm8::Zeros{24});
        when '100' =>
            imm64 = Replicate{64}(Zeros{8}::imm8);
        when '101' =>
            imm64 = Replicate{64}(imm8::Zeros{8});
        when '110' =>
            if cmode[0] == '0' then
                imm64 = Replicate{64}(Zeros{16}::imm8::Ones{8});
            else
                imm64 = Replicate{64}(Zeros{8}::imm8::Ones{16});
            end;
        when '111' =>
            if cmode[0] == '0' && op == '0' then
                imm64 = Replicate{64}(imm8);
            end;
            if cmode[0] == '0' && op == '1' then
                let imm8a : bits(8) = Replicate{}(imm8[7]);
                let imm8b : bits(8) = Replicate{}(imm8[6]);
                let imm8c : bits(8) = Replicate{}(imm8[5]);
                let imm8d : bits(8) = Replicate{}(imm8[4]);
                let imm8e : bits(8) = Replicate{}(imm8[3]);
                let imm8f : bits(8) = Replicate{}(imm8[2]);
                let imm8g : bits(8) = Replicate{}(imm8[1]);
                let imm8h : bits(8) = Replicate{}(imm8[0]);
                imm64 = imm8a::imm8b::imm8c::imm8d::imm8e::imm8f::imm8g::imm8h;
            end;
            if cmode[0] == '1' && op == '0' then
                let imm32 : bits(32) = (imm8[7]::NOT(imm8[6])::Replicate{5}(imm8[6])::
                    imm8[5:0]::Zeros{19});
                imm64 = Replicate{64}(imm32);
            end;
            if cmode[0] == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding(); end;
                imm64 = imm8[7]::NOT(imm8[6])::Replicate{8}(imm8[6])::imm8[5:0]::Zeros{48};
            end;
        end;
    end;

    return imm64;
end;
```

Library pseudocode for shared/functions/vector/MatMulAdd

```
// MatMulAdd()
// =====
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

func MatMulAdd(addend : bits(128), op1 : bits(128), op2 : bits(128), op1_unsigned : boolean,
               op2_unsigned : boolean) => bits(128)
begin
    var result : bits(128);
    var sum : bits(32);
    var prod : integer;

    for i = 0 to 1 do
        for j = 0 to 1 do
            sum = addend[(2*i + j)*:32];
            for k = 0 to 7 do
                let opelt1 : bits(8) = op1[(8*i + k)*:8];
                let opelt2 : bits(8) = op2[(8*j + k)*:8];
                let element1 : integer = if op1_unsigned then UInt(opelt1) else SInt(opelt1);
                let element2 : integer = if op2_unsigned then UInt(opelt2) else SInt(opelt2);
                prod = element1 * element2;
                sum = sum + prod;
            end;
            result[(2*i + j)*:32] = sum;
        end;
    end;

    return result;
end;
```

Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// =====

func PolynomialMult{M, N}(op1 : bits(M), op2 : bits(N)) => bits(M+N)
begin
    var result : bits(N + M) = Zeros{M+N};
    let extended_op2 : bits(N + M) = ZeroExtend{M+N}(op2);
    for i=0 to M-1 do
        if op1[i] == '1' then
            result = result XOR LSL(extended_op2, i);
        end;
    end;
    return result;
end;
```

Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// =====

func SatQ{N}(i : integer, unsigned : boolean) => (bits(N), boolean)
begin
    let (result, sat) : (bits(N), boolean) = (if unsigned then UnsignedSatQ{N}(i)
                                              else SignedSatQ{N}(i));
    return (result, sat);
end;
```

Library pseudocode for shared/functions/vector/ShiftSat

```
// ShiftSat()
// =====

func ShiftSat(shift : integer, esize : integer) => integer
begin
    if shift > esize+1 then return esize+1;
    elsif shift < -(esize+1) then return -(esize+1); end;
    return shift;
end;
```

Library pseudocode for shared/functions/vector/SignedSat

```
// SignedSat()
// =====

func SignedSat{N}(i : integer) => bits(N)
begin
  let (result, -) = SignedSatQ{N}(i);
  return result;
end;
```

Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// =====

func SignedSatQ{N}(i : integer) => (bits(N), boolean)
begin
  var result : integer;
  var saturated : boolean;
  if i > 2^(N-1) - 1 then
    result = 2^(N-1) - 1; saturated = TRUE;
  elsif i < -(2^(N-1)) then
    result = -(2^(N-1)); saturated = TRUE;
  else
    result = i; saturated = FALSE;
  end;
  return (result[N-1:0], saturated);
end;
```

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =====

func UnsignedRSqrtEstimate{N}(operand : bits(N)) => bits(N)
begin
  assert N == 32;
  var result : bits(N);
  if operand[N-1:N-2] == '00' then // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
    result = Ones{N};
  else
    // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
    // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
    let increasedprecision : boolean = FALSE;
    let estimate : integer = RecipSqrtEstimate(UInt(operand[31:23]), increasedprecision);
    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate[8:0] :: Zeros{N-9};
  end;

  return result;
end;
```

Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =====

func UnsignedRecipEstimate{N}(operand : bits(N)) => bits(N)
begin
  assert N == 32;
  var result : bits(N);
  if operand[N-1] == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
    result = Ones{N};
  else
    // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

    // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
    let increasedprecision : boolean = FALSE;
    let estimate : integer = RecipEstimate(UInt(operand[31:23]), increasedprecision);

    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate[8:0] :: Zeros{N-9};
  end;

  return result;
end;
```

Library pseudocode for shared/functions/vector/UnsignedSat

```
// UnsignedSat()
// =====

func UnsignedSat{N}(i : integer) => bits(N)
begin
    let (result, -) = UnsignedSatQ{N}(i);
    return result;
end;
```

Library pseudocode for shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// =====

func UnsignedSatQ{N}(i : integer) => (bits(N), boolean)
begin
    var result : integer;
    var saturated : boolean;
    if i > 2^N - 1 then
        result = 2^N - 1; saturated = TRUE;
    elsif i < 0 then
        result = 0; saturated = TRUE;
    else
        result = i; saturated = FALSE;
    end;
    return (result[N-1:0], saturated);
end;
```

Library pseudocode for shared/trace/Common/DebugMemWrite

```
// DebugMemWrite()
// =====
// Write data to memory one byte at a time. Starting at the passed virtual address.
// Used by SPE and TRBE.

func DebugMemWrite(vaddress : bits(64), accdesc : AccessDescriptor,
    aligned : boolean, data : bits(8)) => (PhysMemRetStatus, AddressDescriptor)
begin
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;

    // Translate virtual address
    var addrdesc : AddressDescriptor;
    let size : integer = 1;
    addrdesc = AArch64 TranslateAddress(vaddress, accdesc, aligned, size);

    if IsFault(addrdesc) then
        return (memstatus, addrdesc);
    end;

    memstatus = PhysMemWrite{8}(addrdesc, accdesc, data);

    return (memstatus, addrdesc);
end;
```



```

// DebugWriteExternalAbort()
// =====
// Populate the syndrome register for an External abort caused by a call of DebugMemWrite().

func DebugWriteExternalAbort(memstatus : PhysMemRetStatus, addrdesc : AddressDescriptor,
                             start_vaddr : bits(64))
begin

    let iswrite : boolean = TRUE;

    var handle_as_SError : boolean = FALSE;
    case addrdesc.fault.accessdesc.acctype of
        when AccessType\_SPE =>
            handle_as_SError = ImpDefBool("Report SPE ExtAbort as SError");
        when AccessType\_TRBE =>
            handle_as_SError = ImpDefBool("Report TRBE ExtAbort as SError");
        otherwise =>
            unreachable;
    end;

    let ttw_abort : boolean = addrdesc.fault.statuscode IN {Fault\_SyncExternalOnWalk,
                                                            Fault\_SyncParityOnWalk};
    let statuscode : Fault = (if ttw_abort then addrdesc.fault.statuscode
                             else memstatus.statuscode);

    if statuscode IN {Fault\_AsyncExternal, Fault\_AsyncParity} || handle_as_SError then
        // Report the abort as an SError
        var fault : FaultRecord = NoFault();
        let parity : boolean = statuscode IN {Fault\_SyncParity, Fault\_AsyncParity,
                                              Fault\_SyncParityOnWalk};
        fault.statuscode = if parity then Fault\_AsyncParity else Fault\_AsyncExternal;
        if IsFeatureImplemented(FEAT_RAS) then
            fault.merrorstate = memstatus.merrorstate;
        end;
        let extflag : bit = if ttw_abort then addrdesc.fault.extflag else memstatus.extflag;
        fault.extflag = extflag;
        fault.accessdesc.acctype = addrdesc.fault.accessdesc.acctype;
        PendSErrorInterrupt(fault);
        return;
    end;

    // Generate a buffer management event, modifying the existing syndrome.
    var handle_async : boolean = FALSE;
    var syndrome : bits(64);
    case addrdesc.fault.accessdesc.acctype of
        when AccessType\_SPE =>
            handle_async = ImpDefBool("Report SPE ExtAbort asynchronously");
            assert !IsFeatureImplemented(FEAT_SPE_EXC);
            syndrome = PMBSR_EL1();
        when AccessType\_TRBE =>
            handle_async = ImpDefBool("Report TRBE ExtAbort asynchronously");
            assert !IsFeatureImplemented(FEAT_TRBE_EXC);
            syndrome = TRBSR_EL1();
        otherwise =>
            unreachable;
    end;

    var ec : bits(6);
    if (IsFeatureImplemented(FEAT_RME) && addrdesc.fault.gpcf.gpf != GPCF\_None &&
        addrdesc.fault.gpcf.gpf != GPCF\_Fail) then
        ec = '011110';
    else
        ec = if addrdesc.fault.secondstage then '100101' else '100100';
    end;

    let mss2 : bits(24) = Zeros{};
    var mss : bits(16) = Zeros{};
    if handle_async then // FSC bits
        mss[5:0] = '010001';
    else
        mss[5:0] = EncodeLDFSC(statuscode, addrdesc.fault.level);
    end;

    // The following values are always updated in the syndrome register.
    if (addrdesc.fault.accessdesc.acctype == AccessType\_SPE &&
        (handle_async || start_vaddr != addrdesc.vaddress)) then
        syndrome[19] = '1'; // DL bit (SPE only)
    end;

    syndrome[18] = '1'; // EA bit

    // The following values are not modified if a previous buffer management event
    // has not been handled. Note that in this simple sequential model, this test
    // will never fail.
    if syndrome[17] == '0' then // Check previous 'S' bit.
        syndrome[55:32] = mss2; // MSS2 bits
        syndrome[31:26] = ec; // EC bits
        if addrdesc.fault.accessdesc.acctype == AccessType\_TRBE then
            syndrome[22] = '1'; // IRQ bit (TRBE only)
        end
    end
end

```

```

    end;
    syndrome[17] = '1';           // S bit
    syndrome[15:0] = mss;         // MSS bits
end;

case addrdesc.fault.accessdesc.acctype of
    when AccessType\_SPE =>
        PMBSR_EL1() = syndrome;
    when AccessType\_TRBE =>
        TRBSR_EL1() = syndrome;
    otherwise =>
        unreachable;
end;
end;

```

Library pseudocode for shared/trace/Common/DebugWriteFault

```
// DebugWriteFault()
// =====
// Populate the syndrome register for a fault caused by a call of DebugMemWrite().

func DebugWriteFault(vaddress : bits(64), fault : FaultRecord)
begin
    var ec : bits(6);
    if (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF\_None &&
        fault.gpcf.gpf != GPCF\_Fail) then
        ec = '011110';
    else
        ec = if fault.secondstage then '100101' else '100100';
    end;

    var mss2 : bits(24) = Zeros{};
    if fault.statuscode == Fault\_Permission then
        mss2[8] = if fault.toplevel then '1' else '0'; // TopLevel bit
        mss2[7] = if fault.assuredonly then '1' else '0'; // AssuredOnly bit
        mss2[6] = if fault.overlay then '1' else '0'; // Overlay bit
        mss2[5] = if fault.dirtybit then '1' else '0'; // DirtyBit
    end;

    var mss : bits(16) = Zeros{};
    if !IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF\_None &&
        fault.gpcf.gpf != GPCF\_Fail) then
        mss[5:0] = EncodeLDFSC(fault.statuscode, fault.level); // FSC bits
    end;

    // Generate a buffer management event, modifying the existing syndrome.
    var target_el : bits(2);
    var syndrome : bits(64);
    case fault.accessdesc.acctype of
        when AccessType\_SPE =>
            target_el = ReportSPEEvent(ec, mss[5:0]);
            syndrome = PMBSR\_EL(target_el);
        when AccessType\_TRBE =>
            target_el = ReportTRBEEEvent(ec, mss[5:0]);
            syndrome = TRBSR\_EL(target_el);
        otherwise =>
            unreachable;
    end;

    // The following values are not modified if a previous buffer management event
    // has not been handled. Note that in this simple sequential model, this test
    // will never fail.
    if syndrome[17] == '0' then // Check previous 'S' bit.
        syndrome[55:32] = mss2; // MSS2 bits
        syndrome[31:26] = ec; // EC bits
        if fault.accessdesc.acctype == AccessType\_TRBE then
            syndrome[22] = '1'; // IRQ bit (TRBE only)
        end;
        syndrome[17] = '1'; // S bit
        syndrome[15:0] = mss; // MSS bits
    end;

    // For SPE, PMBPTR_EL1 points to the address that generated the fault, and writing
    // to memory never started. Therefore, there is no data loss and DL is unchanged.

    case fault.accessdesc.acctype of
        when AccessType\_SPE =>
            PMBSR\_EL(target_el) = syndrome;

        when AccessType\_TRBE =>
            TRBSR\_EL(target_el) = syndrome;

        otherwise =>
            unreachable;
    end;

    return;
end;
```


Library pseudocode for shared/trace/Common/GetTimestamp

```
// GetTimestamp()
// =====
// Returns the Timestamp depending on the type

func GetTimestamp(timestampType : TimeStamp) => bits(64)
begin
  case timestampType of
    when TimeStamp Physical =>
      return PhysicalCountInt();
    when TimeStamp Virtual =>
      return PhysicalCountInt() - CNTVOFF_EL2();
    when TimeStamp OffsetPhysical =>
      let physoff : bits(64) = if PhysicalOffsetIsValid() then CNTPOFF_EL2() else Zeros{64};
      return PhysicalCountInt() - physoff;
    when TimeStamp None =>
      return Zeros{64};
    when TimeStamp CoreSight =>
      return ImpDefBits{64}("CoreSight timestamp");
    otherwise =>
      unreachable;
  end;
end;
```

Library pseudocode for shared/trace/Common/PhysicalOffsetIsValid

```
// PhysicalOffsetIsValid()
// =====
// Returns whether the Physical offset for the timestamp is valid

func PhysicalOffsetIsValid() => boolean
begin
  if !HaveAArch64() then
    return FALSE;
  elsif !HaveEL(EL2) || !IsFeatureImplemented(FEAT_ECV_POFF) then
    return FALSE;
  elsif HaveEL(EL3) && EffectiveSCR\_EL3\_NS() == '1' && EffectiveSCR\_EL3\_RW() == '0' then
    return FALSE;
  elsif HaveEL(EL3) && SCR_EL3().ECVEn == '0' then
    return FALSE;
  elsif CNTHCTL_EL2().ECV == '0' then
    return FALSE;
  else
    return TRUE;
  end;
end;
```

Library pseudocode for shared/trace/Common/TRBCRManStopWrite

```
// TRBCRManStopWrite()
// =====
// Called on a write of 1 to TRBCR.ManStop.

func TRBCRManStopWrite()
begin
  TraceUnitFlush();
  OtherTRBManagementEvent('000011');
  TryAssertTRBIRQ();
end;
```

Library pseudocode for shared/trace/TraceBranch/BranchNotTaken

```
// BranchNotTaken()
// =====
// Called when a branch is not taken.

func BranchNotTaken(branchtype : BranchType, branch_conditional : boolean)
begin
  let branchtaken : boolean = FALSE;
  if IsFeatureImplemented(FEAT_SPE) then
    SPEBranch{64}(ARBITRARY : bits(64), branchtype, branch_conditional, branchtaken);
  end;
  return;
end;
```

Library pseudocode for shared/trace/TraceBuffer/AllowExternalTraceBufferAccess

```
// AllowExternalTraceBufferAccess()
// =====
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed for the given Security state, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

func AllowExternalTraceBufferAccess(addrdesc : AddressDescriptor) => boolean
begin
    assert IsFeatureImplemented(FEAT_TRBE_EXT);
    // FEAT_Debugv8p4 is always implemented when FEAT_TRBE_EXT is implemented.
    assert IsFeatureImplemented(FEAT_Debugv8p4);

    var etbad : bits(2) = if HaveEL\(EL3\) then MDCR_EL3().ETBAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && etbad IN {'01','10'} then
        (-, etbad) = ConstrainUnpredictableBits{2}(Unpredictable RES ETBAD);
        // The value returned by ConstrainUnpredictableBits must be a
        // non-reserved value
        assert etbad IN {'00', '11'};
    end;

    case etbad of
        when '00' =>
            if IsFeatureImplemented(FEAT_RME) then
                return addrdesc.paddress.paspace == PAS Root;
            else
                return addrdesc.paddress.paspace == PAS Secure;
            end;
        when '01' =>
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS Root, PAS Realm};
        when '10' =>
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS Root, PAS Secure};
        when '11' =>
            return TRUE;
    end;
end;
```

Library pseudocode for shared/trace/TraceBuffer/CheckForTRBEEException

```
// CheckForTRBEEException()
// =====
// Take a TRBE Profiling exception if pending, permitted, and unmasked.

func CheckForTRBEEException()
begin
    if !IsFeatureImplemented(FEAT_TRBE_EXC) || !SelfHostedTraceEnabled() then
        return;
    end;

    if Halted() || Restarting() then
        return;
    end;

    var route_to_el3 : boolean = FALSE;
    var route_to_el2 : boolean = FALSE;
    var route_to_el1 : boolean = FALSE;

    if HaveEL(EL3) && MDCR_EL3().TRBEE == '1x' then
        let pending : boolean = TRBSR_EL3().IRQ == '1';
        let masked : boolean = PSTATE.EL == EL3;
        route_to_el3 = pending && !masked;
    end;

    var owning_ss : SecurityState;
    var owning_el : bits(2);
    (owning_ss, owning_el) = TraceBufferOwner();
    let in_owning_ss : boolean = IsCurrentSecurityState(owning_ss);

    if EffectiveTRFCR_EL2_EE() IN {'1x'} then
        let pending : boolean = TRBSR_EL2().IRQ == '1';
        let masked : boolean = (!in_owning_ss || PSTATE.EL == EL3 ||
                                (PSTATE.EL == EL2 && (TRFCR_EL2().EE != '11' ||
                                                         TRFCR_EL2().KE == '0' ||
                                                         PSTATE.PM == '1')));
        route_to_el2 = pending && !masked;
    end;

    if EffectiveTRFCR_EL1_EE() == '11' then
        let pending : boolean = TRBSR_EL1().IRQ == '1';
        let masked : boolean = (!in_owning_ss || PSTATE.EL IN {EL3, EL2} ||
                                (PSTATE.EL == EL1 && (TRFCR_EL1().KE == '0' ||
                                                         PSTATE.PM == '1')));
        if EffectiveTGE() == '1' then
            route_to_el2 = route_to_el2 || (pending && !masked);
        else
            route_to_el1 = pending && !masked;
        end;
    end;

    let fsc : bits(5) = '00010'; // TRBE exception
    let synchronous : boolean = FALSE;

    // The relative priorities of the following checks is IMPLEMENTATION DEFINED
    if route_to_el3 then
        TakeProfilingException(EL3, fsc, synchronous);
    end;
    if route_to_el2 then
        TakeProfilingException(EL2, fsc, synchronous);
    end;
    if route_to_el1 then
        TakeProfilingException(EL1, fsc, synchronous);
    end;
end;
```

Library pseudocode for shared/trace/TraceBuffer/CheckMDCR_EL3_NSTBTrap

```
// CheckMDCR_EL3_NSTBTrap()
// =====
// Check if the register access is trappable by MDCR_EL3.[NSTBE, NSTB]

func CheckMDCR_EL3_NSTBTrap() => boolean
begin
    var state_bits : bits(3);
    var reserved : boolean;
    (state_bits, reserved) = EffectiveMDCR_EL3_NSTB();
    return ((reserved && ConstrainUnpredictableBool(Unpredictable_RESERVEDNSxB_Trp)) ||
            state_bits[0] == '0' || state_bits[1] != EffectiveSCR_EL3_NS() ||
            (IsFeatureImplemented(FEAT_RME) && state_bits[2] != EffectiveSCR_EL3_NSE()));
end;
```



```

// CollectTrace()
// =====
// Called for each byte generated by the trace unit.
// Returns TRUE if the Trace Buffer Unit accepts or discards the trace
// data, and FALSE if the Trace Buffer Unit rejects the trace data.

func CollectTrace(datum : bits(8)) => boolean recurselimit ElementStreamSize
begin
    if !TraceBufferEnabled() then // Trace buffer disabled
        // 'datum' is discarded
        if HaveImpDefTraceOutput() then
            return ImpDefTraceOutput(datum);
        else
            return TRUE; // Discard the trace byte
        end;
    end;

    // If the TRBE cannot accept the trace data, it must return FALSE
    if TRBEInternalBufferFull() then
        return FALSE;
    end;

    if TraceBufferRunning() then // Accept the data
        let address : bits(64) = TRBPTR_EL1();
        var ttw_abort : boolean = FALSE;
        let ttw_abort_as_fault : boolean = (ImpDefBool(
            "Report TRBE ExtAbort on TTW as fault"));

        var addrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        if !SelfHostedTraceEnabled() then
            // The Trace Buffer Unit is using External mode.
            if IsFeatureImplemented(FEAT_RME) && !ExternalRootInvasiveDebugEnabled() then
                if IsZero(GPCCR_EL3().[TBGPCD, GPC]) then
                    return FALSE;
                end;
            end;

            let pas : bits(2) = TRBMAR_EL1().PAS;
            let paspace : PAspace = DecodePAspace('0', pas[1], pas[0]);
            var valid_config : boolean = IsPASValid(pas) && InvasiveDebugPermittedPAS(paspace);
            if IsFeatureImplemented(FEAT_TRBE_MPAM) && TRBMPAM_EL1().EN == '1' then
                let mpam_sp : bits(2) = TRBMPAM_EL1().MPAM_SP;
                let mpam_pa : PAspace = DecodePAspace('0', mpam_sp[1], mpam_sp[0]);
                valid_config = (valid_config && IsPASValid(mpam_sp) &&
                    InvasiveDebugPermittedPAS(mpam_pa));
            end;

            if !valid_config then
                OtherTRBEManagementEvent('000000');
                TryAssertTRBIRQ();
                return TRUE;
            end;

            let el : bits(2) = ARBITRARY : bits(2);
            let ss : SecurityState = ARBITRARY : SecurityState;
            let accdesc : AccessDescriptor = CreateAccDescTRBE(ss, el);

            var pa : FullAddress;
            pa.address = address[NUM_PABITS-1:0];
            pa.paspace = paspace;
            let memattrs : MemoryAttributes = S1DecodeMemAttrs(TRBMAR_EL1().Attr,
                TRBMAR_EL1().SH, TRUE);
            addrdesc = CreateAddressDescriptor(pa, memattrs, accdesc);
            addrdesc.mecid = DEFAULT_MECID;
            if IsFeatureImplemented(FEAT_RME) && !ExternalRootInvasiveDebugEnabled() then
                let gpcf : GPCFRecord = GranuleProtectionCheck(addrdesc, accdesc);
                if gpcf.gpcf == GPCF_None then
                    memstatus = PhysMemWrite{8}(addrdesc, accdesc, datum);
                else
                    addrdesc.fault.gpcf = gpcf;
                    addrdesc.fault.statuscode = Fault_GPCFOnOutput;
                end;
            else
                memstatus = PhysMemWrite{8}(addrdesc, accdesc, datum);
            end;
        else
            // The Trace Buffer Unit is using Self-hosted mode.
            var owning_ss : SecurityState;
            var owning_el : bits(2);
            (owning_ss, owning_el) = TraceBufferOwner();
            let accdesc : AccessDescriptor = CreateAccDescTRBE(owning_ss, owning_el);

            let aligned : boolean = TRUE;
            (memstatus, addrdesc) = DebugMemWrite(address, accdesc, aligned, datum);

            ttw_abort = addrdesc.fault.statuscode IN {Fault_SyncExternalOnWalk,
                Fault_SyncParityOnWalk};
        end;
    end;
end;

```

```

if IsFault(addrdesc.fault.statuscode) && (!ttw_abort || ttw_abort_as_fault) then
    DebugWriteFault(address, addrdesc.fault);
    TryAssertTRBIRQ();
    return TRUE;
elsif IsFault(memstatus) || (ttw_abort && !ttw_abort_as_fault) then
    DebugWriteExternalAbort(memstatus, addrdesc, address);
    TryAssertTRBIRQ();
    return TRUE;
end;

// Check for Trigger Event
let target_el : bits(2) = DefaultTRBEEEvent();
let triggered : boolean = TRBSR_EL(target_el).TRG == '1';
if triggered && !IsZero(TRBTRG_EL1().TRG) then
    TRBTRG_EL1().TRG = (TRBTRG_EL1().TRG - 1)[31:0];
    if IsZero(TRBTRG_EL1().TRG) && TRBLIMITR_EL1().TM != '11' then
        TraceUnitFlush();
        TraceUnitFlushOnTriggerComplete();
    end;
end;

// Increment the pointer
var next_address : bits(64) = TRBPTR_EL1() + 1;
if next_address[63:12] == TRBLIMITR_EL1().LIMIT then
    next_address = TRBBASER_EL1().BASE::Zeros{12};
    TRBSR_EL(target_el).WRAP = '1';
    CTI_SignalEvent(CrossTriggerIn TRBEWrap);
    if TRBLIMITR_EL1().FM == '00' then // Fill mode
        let bsc : bits(6) = '000001'; // Buffer full event
        OtherTRBEManagementEvent(bsc);
    elsif TRBLIMITR_EL1().FM != '11' then // Not Circular Buffer mode
        if TRBSR_EL(target_el).IRQ == '0' then
            TRBSR_EL(target_el).IRQ = '1'; // Assert interrupt or exception
            CTI_SignalEvent(CrossTriggerIn TRBEMgmt);
        end;
    end;
end;

end;
TRBPTR_EL1() = next_address[63:0];

TryAssertTRBIRQ();
end;
return TRUE;
end;

```

Library pseudocode for shared/trace/TraceBuffer/DefaultTRBEEEvent

```

// DefaultTRBEEEvent()
// =====
// Return the target ELx for an indirect write to TRBSR_ELx for an Other buffer management
// event or anything other than a buffer management event.

func DefaultTRBEEEvent() => bits(2)
begin
    return ReportTRBEEEvent(Zeros{6}, ARBITRARY : bits(6));
end;

```

Library pseudocode for shared/trace/TraceBuffer/DetectedTraceTrigger

```

// DetectedTraceTrigger()
// =====
// Called when the trace unit detects a trace trigger

func DetectedTraceTrigger()
begin
    if TraceBufferRunning() then
        let target_el : bits(2) = DefaultTRBEEEvent();
        if TRBSR_EL(target_el).TRG == '0' then
            TRBSR_EL(target_el).TRG = '1';
            if IsZero(TRBTRG_EL1().TRG) && TRBLIMITR_EL1().TM != '11' then
                TraceUnitFlush();
                TraceUnitFlushOnTriggerComplete();
            end;
        end;
    end;
end;
end;

```

Library pseudocode for shared/trace/TraceBuffer/EffectiveMDCR_EL3_NSTB

```
// EffectiveMDCR_EL3_NSTB()
// =====
// Return the Effective value of MDCR_EL3().[NSTBE, NSTB] field and whether it is a reserved value.

func EffectiveMDCR_EL3_NSTB() => (bits(3), boolean)
begin
    var state_bits : bits(3);
    var reserved    : boolean = FALSE;

    if IsFeatureImplemented(FEAT_RME) then
        state_bits = MDCR_EL3().[NSTBE, NSTB];
        if state_bits == '10x' || (!IsFeatureImplemented(FEAT_Secure) && state_bits == '00x') then
            // Reserved value
            reserved = TRUE;
            (-, state_bits) = ConstrainUnpredictableBits{3}(Unpredictable_RESERVEDNSxB);
        end;
    else
        state_bits = '0' :: MDCR_EL3().NSTB;
    end;

    return (state_bits, reserved);
end;
```

Library pseudocode for shared/trace/TraceBuffer/EffectiveTRBLIMITR_EL1_nVM

```
// EffectiveTRBLIMITR_EL1_nVM()
// =====

func EffectiveTRBLIMITR_EL1_nVM() => bit
begin
    if !SelfHostedTraceEnabled() then
        // If SelfHostedTraceEnabled() is FALSE, then this function is only called when
        // FEAT_TRBE_EXT is implemented.
        assert IsFeatureImplemented(FEAT_TRBE_EXT);
        return '1';
    end;
    if IsFeatureImplemented(FEAT_TRBEv1p1) && HaveEL(EL2) then
        let (owning_ss, owning_el) = TraceBufferOwner();
        if ((owning_ss != SS_Secure || IsSecureEL2Enabled()) && owning_el == EL1 &&
            TRFCR_EL2().DnVM == '1') then
            return '0';
        end;
    end;
    return TRBLIMITR_EL1().nVM;
end;
```

Library pseudocode for shared/trace/TraceBuffer/EffectiveTRFCR_EL1_EE

```
// EffectiveTRFCR_EL1_EE()
// =====
// Return the Effective value of TRFCR_EL1.EE for the purpose of controlling the
// TRBE Profiling exception.

func EffectiveTRFCR_EL1_EE() => bits(2)
begin
    if EffectiveTRFCR_EL2_EE() == '00' then
        return '00';
    end;

    var ee : bits(2) = TRFCR_EL1().EE;
    if ee IN {'01', '10'} then // Reserved value
        if IsFeatureImplemented(FEAT_NV) then
            ee[0] = ee[1];
        else
            var c : Constraint;
            (c, ee) = ConstrainUnpredictableBits{2}(Unpredictable_RESTRFEE);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then
                ee = '00';
            end;
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        end;
    end;

    return ee;
end;
```

Library pseudocode for shared/trace/TraceBuffer/EffectiveTRFCR_EL2_EE

```
// EffectiveTRFCR_EL2_EE()
// =====
// Return the Effective value of TRFCR_EL2.EE.

func EffectiveTRFCR_EL2_EE() => bits(2)
begin
    if !IsFeatureImplemented(FEAT_TRBE_EXC) || !SelfHostedTraceEnabled() then
        return '00';
    end;

    if HaveEL(EL3) && MDCR_EL3().TRBEE == '00' then
        return '00';
    end;

    let check_el2 : boolean = HaveEL(EL2) && (EffectiveSCR_EL3_NS() == '1' ||
        IsSecureEL2Enabled());
    return if check_el2 then TRFCR_EL2().EE else '01';
end;
```

Library pseudocode for shared/trace/TraceBuffer/ElementStreamSize

```
// ElementStreamSize
// =====

config ElementStreamSize : integer{0..4096} = 4096;
```

Library pseudocode for shared/trace/TraceBuffer/GetTRBSR_EL1_FSC

```
// GetTRBSR_EL1_FSC()
// =====
// Query the TRBSR_EL1.FSC field.

func GetTRBSR_EL1_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = TRBSR_EL1()[5:0];
    return FSC;
end;
```

Library pseudocode for shared/trace/TraceBuffer/GetTRBSR_EL2_FSC

```
// GetTRBSR_EL2_FSC()
// =====
// Query the TRBSR_EL2.FSC field.

func GetTRBSR_EL2_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = TRBSR_EL2()[5:0];
    return FSC;
end;
```

Library pseudocode for shared/trace/TraceBuffer/GetTRBSR_EL3_FSC

```
// GetTRBSR_EL3_FSC()
// =====
// Query the TRBSR_EL3.FSC field.

func GetTRBSR_EL3_FSC() => bits(6)
begin
    var FSC : bits(6);

    FSC = TRBSR_EL3()[5:0];
    return FSC;
end;
```

Library pseudocode for shared/trace/TraceBuffer/HaveImpDefTraceOutput

```
// HaveImpDefTraceOutput()
// =====

func HaveImpDefTraceOutput() => boolean
begin
    return ImpDefBool("Has Enabled External Trace Port");
end;
```


Library pseudocode for shared/trace/TraceBuffer/ImpDefTraceOutput

```
// ImpDefTraceOutput()
// =====

func ImpDefTraceOutput(datum : bits(8)) => boolean
begin
    // Send 'datum' to an IMPLEMENTATION DEFINED trace output port
    // return TRUE if the byte is sent
    return FALSE;
end;
```

Library pseudocode for shared/trace/TraceBuffer/OtherTRBManagementEvent

```
// OtherTRBManagementEvent()
// =====
// Report an Other buffer management event, with the status code 'bsc'

func OtherTRBManagementEvent(bsc : bits(6))
begin
    ReportTRBManagementEvent('000000', bsc);
end;
```

Library pseudocode for shared/trace/TraceBuffer/ReportTRBEEEvent

```
// ReportTRBEEEvent()
// =====
// Return the target ELx for an indirect write to TRBSR_ELx.
// When the indirect write is due to a buffer management event:
// 'ec_bits' is the Event Class for the management event.
// 'fsc_bits' is the Fault Status Code when this is a fault, ignored otherwise.
// Otherwise, 'ec_bits' should be Zeros().

func ReportTRBEEEvent(ec_bits : bits(6), fsc_bits : bits(6)) => bits(2)
begin
    var target_el : bits(2);
    var route_to_el3 : boolean = FALSE;
    var route_to_el2 : boolean = FALSE;

    if IsFeatureImplemented(FEAT_TRBE_EXC) && SelfHostedTraceEnabled() then
        if ec_bits == '011111' then // implementation defined fault
            return ImpDefBits{2}("target_el for TRBE ImpDef Fault");
        end;

        let s1fault : boolean = (ec_bits == '100100'); // Stage 1 fault
        let s2fault : boolean = (ec_bits == '100101'); // Stage 2 fault

        var gpcfault, gpfault : boolean;
        if IsFeatureImplemented(FEAT_RME) then
            // Granule Protection Check fault, other than GPF. That is, a GPT address size fault,
            // GPT walk fault, or synchronous External abort on GPT fetch.
            gpcfault = (ec_bits == '011110');
            // Other Granule Protection Fault, reported as Stage 1 or Stage 2 fault.
            gpfault = ((s1fault || s2fault) && fsc_bits IN {'10001x', '1001xx', '101000'});
        else
            gpcfault = FALSE;
            gpfault = FALSE;
        end;
        let sync_ext_abort : boolean = ((s1fault || s2fault) &&
            fsc_bits IN {'010000', '01001x', '0101xx', '011011'});

        var owning_ss : SecurityState;
        var owning_el : bits(2);
        (owning_ss, owning_el) = TraceBufferOwner();

        if HaveEL(EL3) && MDCR_EL3().TRBEE == '1x' then
            route_to_el3 = (MDCR_EL3().TRBEE == '11' ||
                gpcfault || (gpfault && SCR_EL3().GPF == '1') ||
                (sync_ext_abort && EffectiveEA() == '1'));
        end;

        if EffectiveTRFCR_EL2_EE() == '1x' then
            route_to_el2 = (TRFCR_EL2().EE == '11' || (s1fault && owning_el == EL2) || s2fault ||
                gpcfault || (gpfault && HCR_EL2().GPF == '1') ||
                (sync_ext_abort && EffectiveHCR_TEA() == '1'));
        end;
    end;

    if route_to_el3 then
        target_el = EL3;
    elseif route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;
    end;

    return target_el;
end;
```

Library pseudocode for shared/trace/TraceBuffer/ReportTRBEManagementEvent

```
// ReportTRBEManagementEvent()
// =====
// Report a buffer management event with the event class 'ec' and status code 'bsc'

func ReportTRBEManagementEvent(ec : bits(6), bsc : bits(6))
begin
    let target_el : bits(2) = DefaultTRBEEEvent();
    if TRBSR_EL(target_el).S == '0' then
        TRBSR_EL(target_el).S = '1'; // Stop collection
    if TRBSR_EL(target_el).IRQ == '0' then
        TRBSR_EL(target_el).IRQ = '1'; // Assert interrupt or exception
        CTI_SignalEvent(CrossTriggerIn_TRBEMgmt);
    end;
    TRBSR_EL(target_el).EC = ec;
    TRBSR_EL(target_el).MSS = ZeroExtend{16}(bsc);
    TRBSR_EL(target_el).MSS2 = Zeros{24};
    CTI_SignalEvent(CrossTriggerIn_TRBESop);
end;
end;
```

Library pseudocode for shared/trace/TraceBuffer/TRBEInternalBufferFull

```
// TRBEInternalBufferFull()
// =====

func TRBEInternalBufferFull() => boolean
begin
    // In the simple sequential model, the internal buffer never fills
    return FALSE;
end;
```

Library pseudocode for shared/trace/TraceBuffer/TRBEInterruptEnabled

```
// TRBEInterruptEnabled()
// =====
// Return TRUE if the TRBE interrupt request (TRBIRQ) is enabled, FALSE otherwise.

func TRBEInterruptEnabled() => boolean
begin
    return EffectiveTRFCR\_EL1\_EE\(\) == '00';
end;
```

Library pseudocode for shared/trace/TraceBuffer/TRBE_TRBIDR_P_Read

```
// TRBE_TRBIDR_P_Read()
// =====
// Called when TRBIDR_EL1 is read, returns the value of TRBIDR_EL1.P

func TRBE_TRBIDR_P_Read() => bit
begin
    var owning_ss : SecurityState;
    var owning_el : bits(2);
    (owning_ss, owning_el) = TraceBufferOwner\(\);

    // Reads as one if the Trace Buffer is owned by a higher Exception
    // Level or another Security state.
    if (UInt(owning_el) > UInt(PSTATE.EL) ||
        (PSTATE.EL != EL3 && owning_ss != CurrentSecurityState\(\))) then
        return '1';
    else
        return '0';
    end;
end;
```

Library pseudocode for shared/trace/TraceBuffer/TRBSR_EL

```
// TRBSR_EL - accessor
// =====

accessor TRBSR_EL(el : bits(2)) <=> value : TRBSRType
begin
    getter
        var r : bits(64);
        case el of
            when EL1 => r = TRBSR_EL1();
            when EL2 => r = TRBSR_EL2();
            when EL3 => r = TRBSR_EL3();
            otherwise => unreachable;
        end;
        return r;
    end;

    setter
        let r : bits(64) = value;
        case el of
            when EL1 => TRBSR_EL1() = r;
            when EL2 => TRBSR_EL2() = r;
            when EL3 => TRBSR_EL3() = r;
            otherwise => unreachable;
        end;
        return;
    end;
end;
```

Library pseudocode for shared/trace/TraceBuffer/TraceBufferEnabled

```
// TraceBufferEnabled()
// =====

func TraceBufferEnabled() => boolean
begin
    if !IsFeatureImplemented(FEAT_TRBE) then
        return FALSE;
    elseif SelfHostedTraceEnabled() then
        if TRBLIMITR_EL1().E == '0' then
            return FALSE;
        end;
        var el : bits(2);
        (-, el) = TraceBufferOwner();
        return !ELUsingAArch32(el);
    elseif IsFeatureImplemented(FEAT_TRBE_EXT) then
        return TRBLIMITR_EL1().XE == '1';
    else
        return FALSE;
    end;
end;
```

Library pseudocode for shared/trace/TraceBuffer/TraceBufferOwner

```
// TraceBufferOwner()
// =====
// Return the owning Security state and Exception level. Must only be called
// when SelfHostedTraceEnabled() is TRUE.

func TraceBufferOwner() => (SecurityState, bits(2))
begin
    assert IsFeatureImplemented(FEAT_TRBE);

    var owning_ss : SecurityState;
    var state_bits : bits(3);
    if HaveEL(EL3) then
        (state_bits, -) = EffectiveMDCR_EL3 NSTB();
    else
        state_bits = if SecureOnlyImplementation() then '001' else '011';
    end;

    case state_bits of
        when '00x' => owning_ss = SS Secure;
        when '01x' => owning_ss = SS NonSecure;
        when '11x' => owning_ss = SS Realm;
    end;
    var owning_el : bits(2);
    if HaveEL(EL2) && (owning_ss != SS Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2().E2TB == '00' then EL2 else EL1;
    else
        owning_el = EL1;
    end;
    return (owning_ss, owning_el);
end;
```

Library pseudocode for shared/trace/TraceBuffer/TraceBufferRunning

```
// TraceBufferRunning()
// =====

func TraceBufferRunning() => boolean
begin
    if !TraceBufferEnabled() then
        return FALSE;
    end;

    var stopped : boolean = TRBSR_EL1().S == '1';
    if IsFeatureImplemented(FEAT_TRBE_EXC) && SelfHostedTraceEnabled() then
        if HaveEL(EL3) && MDCR_EL3().TRBEE == '1x' then
            stopped = stopped || (TRBSR_EL3().S == '1');
        end;
        if EffectiveTRFCR_EL2_EE() == '1x' then
            stopped = stopped || (TRBSR_EL2().S == '1');
        end;
    end;
    return !stopped;
end;
```

Library pseudocode for shared/trace/TraceBuffer/TraceUnitFlushOnTriggerComplete

```
// TraceUnitFlushOnTriggerComplete()
// =====
// Called when a trace unit flush completes following a call to
// TraceUnitFlush() due to a trace trigger.

func TraceUnitFlushOnTriggerComplete()
begin
    if TRBLIMITR_EL1().TM == '00' then      // Stop on trigger
        let bsc : bits(6) = '000010';      // Trigger event
        OtherTRBEManagementEvent(bsc);
    elsif TRBLIMITR_EL1().TM != '11' then    // Not Ignore trigger
        let target_el : bits(2) = DefaultTRBEEEvent();
        TRBSR\_EL(target_el).IRQ = '1';      // Assert interrupt or exception
    end;
end;
```

Library pseudocode for shared/trace/TraceBuffer/TryAssertTRBIRQ

```
// TryAssertTRBIRQ()
// =====
// Assert TRBIRQ pin when appropriate.

func TryAssertTRBIRQ()
begin
    if TRBEInterruptEnabled() && TRBSR_EL1().IRQ == '1' then
        SetInterruptRequestLevel(InterruptID\_TRBIRQ, HIGH);
    else
        SetInterruptRequestLevel(InterruptID\_TRBIRQ, LOW);
    end;
    return;
end;
```

Library pseudocode for shared/trace/TraceInstrumentationAllowed/TraceInstrumentationAllowed

```
// TraceInstrumentationAllowed()
// =====
// Returns TRUE if Instrumentation Trace is allowed
// in the given Exception level and Security state.

func TraceInstrumentationAllowed(ss : SecurityState, el : bits(2)) => boolean
begin
    if !IsFeatureImplemented(FEAT ITE) then return FALSE; end;
    if ELUsingAArch32(el) then return FALSE; end;

    if TraceAllowed(el) then
        var ite_bit : bit;
        case el of
            when EL3 => ite_bit = '0';
            when EL2 => ite_bit = TRCITECR_EL2().E2E;
            when EL1 => ite_bit = TRCITECR_EL1().E1E;
            when EL0 =>
                if EffectiveTGE() == '1' then
                    ite_bit = TRCITECR_EL2().E0HE;
                else
                    ite_bit = TRCITECR_EL1().E0E;
                end;
        end;

        if SelfHostedTraceEnabled() then
            return ite_bit == '1';
        else
            var el_bit : bit;
            var ss_bit : bit;
            case el of
                when EL0 => el_bit = TRCITEEDCR().E0;
                when EL1 => el_bit = TRCITEEDCR().E1;
                when EL2 => el_bit = TRCITEEDCR().E2;
                when EL3 => el_bit = TRCITEEDCR().E3;
            end;
            case ss of
                when SS Realm => ss_bit = TRCITEEDCR().RL;
                when SS Secure => ss_bit = TRCITEEDCR().S;
                when SS NonSecure => ss_bit = TRCITEEDCR().NS;
                otherwise => ss_bit = '1';
            end;

            let ed_allowed : boolean = ss_bit == '1' && el_bit == '1';

            if TRCCONFIGR().ITO == '1' then
                return ed_allowed;
            else
                return ed_allowed && ite_bit == '1';
            end;
        end;
    else
        return FALSE;
    end;
end;
```

Library pseudocode for shared/trace/TraceProcessElements/TraceUnitFlush

```
// TraceUnitFlush()
// =====
// Called when a trace unit flush is requested, to output previous recorded trace.

impdef func TraceUnitFlush()
begin
    return;
end;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE0HTRE

```
// EffectiveE0HTRE()
// =====
// Returns effective E0HTRE value

func EffectiveE0HTRE() => bit
begin
    return if ELUsingAArch32(EL2) then HTRFCR().E0HTRE else TRFCR_EL2().E0HTRE;
end;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE0TRE

```
// EffectiveE0TRE()
// =====
// Returns effective E0TRE value

func EffectiveE0TRE() => bit
begin
    return if ELUsingAArch32\(EL1\) then TRFCR().E0TRE else TRFCR_EL1().E0TRE;
end;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE1TRE

```
// EffectiveE1TRE()
// =====
// Returns effective E1TRE value

func EffectiveE1TRE() => bit
begin
    return if UsingAArch32\(\) then TRFCR().E1TRE else TRFCR_EL1().E1TRE;
end;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE2TRE

```
// EffectiveE2TRE()
// =====
// Returns effective E2TRE value

func EffectiveE2TRE() => bit
begin
    return if UsingAArch32\(\) then HTRFCR().E2TRE else TRFCR_EL2().E2TRE;
end;
```

Library pseudocode for shared/trace/selfhosted/SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// =====
// Returns TRUE if Self-hosted Trace is enabled.

readonly func SelfHostedTraceEnabled() => boolean
begin
    var secure_trace_enable : bit = '0';
    if !(HaveTraceExt\(\) && IsFeatureImplemented(FEAT_TRF)) then return FALSE; end;
    if EDSCR().TFO == '0' then return TRUE; end;
    if IsFeatureImplemented(FEAT_RME) then
        secure_trace_enable = if IsFeatureImplemented(FEAT_SEL2) then MDCR_EL3().STE else '0';
        return ((secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled\(\)) ||
            (MDCR_EL3().RLTE == '1' && !ExternalRealmNoninvasiveDebugEnabled\(\)));
    end;
    if HaveEL\(EL3\) then
        secure_trace_enable = if ELUsingAArch32\(EL3\) then SDCR().STE else MDCR_EL3().STE;
    else
        secure_trace_enable = if SecureOnlyImplementation\(\) then '1' else '0';
    end;

    if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled\(\) then
        return TRUE;
    end;

    return FALSE;
end;
```

Library pseudocode for shared/trace/selfhosted/TraceAllowed

```
// TraceAllowed()
// =====
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

func TraceAllowed(el : bits(2)) => boolean
begin
    if !HaveTraceExt() then
        return FALSE;
    end;
    // If in Debug state then tracing is not allowed
    if Halted() && !Restarting() then
        return FALSE;
    end;
    if SelfHostedTraceEnabled() then
        var trace_allowed : boolean;
        let ss : SecurityState = SecurityStateAtEL(el);
        // Detect scenarios where tracing in this Security state is never allowed.
        case ss of
            when SS_NonSecure =>
                trace_allowed = TRUE;
            when SS_Secure =>
                var trace_bit : bit;
                if HaveEL(EL3) then
                    trace_bit = if ELUsingAArch32(EL3) then SDCR().STE else MDCR_EL3().STE;
                else
                    trace_bit = '1';
                end;
                trace_allowed = trace_bit == '1';
            when SS_Realm =>
                trace_allowed = MDCR_EL3().RLTE == '1';
            when SS_Root =>
                trace_allowed = FALSE;
        end;

        // Tracing is prohibited if the trace buffer owning security state is not the
        // current Security state or the owning Exception level is a lower Exception level.
        if IsFeatureImplemented(FEAT_TRBE) && TraceBufferEnabled() then
            let (owning_ss, owning_el) = TraceBufferOwner();
            if (ss != owning_ss || UInt(owning_el) < UInt(el) ||
                (EffectiveTGE() == '1' && owning_el == EL1)) then
                trace_allowed = FALSE;
            end;
        end;

        var TRE_bit : bit;
        case el of
            when EL3 => TRE_bit = if !HaveAArch64() then TRFCR().E1TRE else '0';
            when EL2 => TRE_bit = EffectiveE2TRE();
            when EL1 => TRE_bit = EffectiveE1TRE();
            when EL0 =>
                if EffectiveTGE() == '1' then
                    TRE_bit = EffectiveEOHTRE();
                else
                    TRE_bit = EffectiveE0TRE();
                end;
        end;

        return trace_allowed && TRE_bit == '1';
    else
        return ExternalNoninvasiveDebugAllowed(el);
    end;
end;
```

Library pseudocode for shared/trace/selfhosted/TraceContextIDR2

```
// TraceContextIDR2()
// =====

func TraceContextIDR2() => boolean
begin
    if !TraceAllowed(PSTATE.EL) || !HaveEL(EL2) then return FALSE; end;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2().CX == '1');
end;
```


Library pseudocode for shared/trace/selfhosted/TraceSynchronizationBarrier

```
// TraceSynchronizationBarrier()
// =====
// Barrier instruction that preserves the relative order of accesses to System
// registers due to trace operations and other accesses to the same registers.
// When FEAT_TRBE is implemented, a TraceSynchronizationBarrier also acts as a memory
// barrier operation to flush any trace data generated by the trace unit, such that
// a subsequent Data Synchronization Barrier does not complete until the trace data
// has been written to memory.

func TraceSynchronizationBarrier()
begin
    if IsFeatureImplemented(FEAT_TRBE) && !TraceAllowed(PSTATE.EL) then
        TraceUnitFlush();
    end;
    return;
end;
```

Library pseudocode for shared/trace/selfhosted/TraceTimeStamp

```
// TraceTimeStamp()
// =====

func TraceTimeStamp() => TimeStamp
begin
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            var TS_el2 : bits(2) = TRFCR_EL2().TS;
            if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits{2}(Unpredictable_EL2TIMESTAMP);
            end;

            case TS_el2 of
                when '00' =>
                    // Falls out to check TRFCR_EL1.TS
                    pass;
                when '01' =>
                    return TimeStamp_Virtual;
                when '10' =>
                    // Otherwise ConstrainUnpredictableBits removes this case
                    assert IsFeatureImplemented(FEAT_ECV);
                    return TimeStamp_OffsetPhysical;
                when '11' =>
                    return TimeStamp_Physical;
            end;
        end;

        var TS_el1 : bits(2) = TRFCR_EL1().TS;
        if TS_el1 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_el1 == '10') then
            // Reserved value
            (-, TS_el1) = ConstrainUnpredictableBits{2}(Unpredictable_EL1TIMESTAMP);
        end;

        case TS_el1 of
            when '01' =>
                return TimeStamp_Virtual;
            when '10' =>
                assert IsFeatureImplemented(FEAT_ECV);
                return TimeStamp_OffsetPhysical;
            when '11' =>
                return TimeStamp_Physical;
            otherwise =>
                unreachable; // ConstrainUnpredictableBits removes this case
        end;
    else
        return TimeStamp_CoreSight;
    end;
end;
```

Library pseudocode for shared/trace/system/IsTraceCorePowered

```
// IsTraceCorePowered()
// =====
// Returns TRUE if the trace unit Core power domain is powered up

impdef func IsTraceCorePowered() => boolean
begin
    return TRUE;
end;
```

Library pseudocode for shared/translation/at

```
type TranslationStage of enumeration {
    TranslationStage_1,
    TranslationStage_12
};

type ATAccess of enumeration {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_Any,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
```

Library pseudocode for shared/translation/at/EncodePARAttrs

```
// EncodePARAttrs()
// =====
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

func EncodePARAttrs(memattrs : MemoryAttributes) => bits(8)
begin
    var result : bits(8);

    if IsFeatureImplemented(FEAT_MTE) && memattrs.tags == MemTag\_AllocationTagged then
        if IsFeatureImplemented(FEAT_MTE_PERM) && memattrs.notagaccess then
            result[7:0] = '11100000';
        else
            result[7:0] = '11110000';
        end;
        return result;
    end;

    if memattrs.memtype == MemType\_Device then
        result[7:4] = '0000';
        case memattrs.device of
            when DeviceType\_nGnRnE => result[3:0] = '0000';
            when DeviceType\_nGnRE => result[3:0] = '0100';
            when DeviceType\_nGRE => result[3:0] = '1000';
            when DeviceType\_GRE => result[3:0] = '1100';
            otherwise => unreachable;
        end;
        result[0] = NOT memattrs.xs;
    else
        if memattrs.xs == '0' then
            if (memattrs.outer.attrs == MemAttr\_WT && memattrs.inner.attrs == MemAttr\_WT &&
                !memattrs.outer.transient && memattrs.outer.hints == MemHint\_RA) then
                return '10100000';
            elsif memattrs.outer.attrs == MemAttr\_NC && memattrs.inner.attrs == MemAttr\_NC then
                return '01000000';
            end;
        end;

        if memattrs.outer.attrs == MemAttr\_WT then
            result[7:6] = if memattrs.outer.transient then '00' else '10';
            result[5:4] = memattrs.outer.hints;
        elsif memattrs.outer.attrs == MemAttr\_WB then
            result[7:6] = if memattrs.outer.transient then '01' else '11';
            result[5:4] = memattrs.outer.hints;
        else // MemAttr_NC
            result[7:4] = '0100';
        end;

        if memattrs.inner.attrs == MemAttr\_WT then
            result[3:2] = if memattrs.inner.transient then '00' else '10';
            result[1:0] = memattrs.inner.hints;
        elsif memattrs.inner.attrs == MemAttr\_WB then
            result[3:2] = if memattrs.inner.transient then '01' else '11';
            result[1:0] = memattrs.inner.hints;
        else // MemAttr_NC
            result[3:0] = '0100';
        end;
    end;

    return result;
end;
```

Library pseudocode for shared/translation/at/PAREncodeShareability

```
// PAREncodeShareability()
// =====
// Derive 64-bit PAR SH field.

func PAREncodeShareability(memattrs : MemoryAttributes) => bits(2)
begin
    if (memattrs.memtype == MemType\_Device ||
        (memattrs.inner.attrs == MemAttr\_NC &&
         memattrs.outer.attrs == MemAttr\_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';
    end;

    case memattrs.shareability of
        when Shareability\_NSH => return '00';
        when Shareability\_ISH => return '11';
        when Shareability\_OSH => return '10';
    end;
end;
```

Library pseudocode for shared/translation/at/ReportedPARAttrs

```
// ReportedPARAttrs()
// =====
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

impdef func ReportedPARAttrs(parattrs : bits(8)) => bits(8)
begin
    return parattrs;
end;
```

Library pseudocode for shared/translation/at/ReportedPARShareability

```
// ReportedPARShareability()
// =====
// The value returned in SH field can be the resulting attribute, as determined by any
// permitted implementation choices and any applicable configuration bits, instead of
// the value that appears in the translation table descriptor.

impdef func ReportedPARShareability(sh : bits(2)) => bits(2)
begin
    return sh;
end;
```

Library pseudocode for shared/translation/attrs/DecodeDevice

```
// DecodeDevice()
// =====
// Decode output Device type

func DecodeDevice(device : bits(2)) => DeviceType
begin
    case device of
        when '00' => return DeviceType\_nGnRnE;
        when '01' => return DeviceType\_nGnRE;
        when '10' => return DeviceType\_nGRE;
        when '11' => return DeviceType\_GRE;
    end;
end;
```

Library pseudocode for shared/translation/atrrs/DecodeLDFAttr

```
// DecodeLDFAttr()
// =====
// Decode memory attributes using LDF (Long Descriptor Format) mapping

func DecodeLDFAttr(attr : bits(4)) => MemAttrHints
begin
    var ldffattr : MemAttrHints;

    if attr == 'x0xx' then ldffattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldffattr.attrs = MemAttr_NC; // Non-cacheable
    elsif attr == 'x1xx' then ldffattr.attrs = MemAttr_WB; // Write-back
    else
        unreachable;
    end;

    // Allocation hints are applicable only to cacheable memory.
    if ldffattr.attrs != MemAttr_NC then
        case attr[1:0] of
            when '00' => ldffattr.hints = MemHint_No; // No allocation hints
            when '01' => ldffattr.hints = MemHint_WA; // Write-allocate
            when '10' => ldffattr.hints = MemHint_RA; // Read-allocate
            when '11' => ldffattr.hints = MemHint_RWA; // Read/Write allocate
        end;
    end;

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldffattr.attrs != MemAttr_NC && ldffattr.hints != MemHint_No then
        ldffattr.transient = attr[3] == '0';
    end;

    return ldffattr;
end;
```

Library pseudocode for shared/translation/atrrs/DecodeSDFAttr

```
// DecodeSDFAttr()
// =====
// Decode memory attributes using SDF (Short Descriptor Format) mapping

func DecodeSDFAttr(rgn : bits(2)) => MemAttrHints
begin
    var sdfattr : MemAttrHints;

    case rgn of
        when '00' => // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr_NC;
        when '01' => // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RWA;
        when '10' => // Write-through, Read allocate
            sdfattr.attrs = MemAttr_WT;
            sdfattr.hints = MemHint_RA;
        when '11' => // Write-back, Read allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RA;
    end;

    sdfattr.transient = FALSE;

    return sdfattr;
end;
```

Library pseudocode for shared/translation/atrrs/DecodeShareability

```
// DecodeShareability()
// =====
// Decode shareability of target memory region

func DecodeShareability(sh : bits(2)) => Shareability
begin
    case sh of
        when '10' => return Shareability_OSH;
        when '11' => return Shareability_ISH;
        when '00' => return Shareability_NSH;
        otherwise =>
            case ConstraintUnpredictable(Unpredictable Shareability) of
                when Constraint_OSH => return Shareability_OSH;
                when Constraint_ISH => return Shareability_ISH;
                when Constraint_NSH => return Shareability_NSH;
            end;
        end;
    end;
end;
```

Library pseudocode for shared/translation/atrrs/EffectiveShareability

```
// EffectiveShareability()
// =====
// Force Outer Shareability on Device and Normal iNCoNC memory

func EffectiveShareability(memattrs : MemoryAttributes) => Shareability
begin
    if (memattrs.memtype == MemType\_Device ||
        (memattrs.inner.attrs == MemAttr\_NC &&
         memattrs.outer.attrs == MemAttr\_NC)) then
        return Shareability\_OSH;
    else
        return memattrs.shareability;
    end;
end;
```

Library pseudocode for shared/translation/atrrs/IsTaggableMemAttr

```
// IsTaggableMemAttr()
// =====
// Determine whether the current memory attributes support MTE

func IsTaggableMemAttr(attrs : MemoryAttributes) => boolean
begin
    return ((attrs.memtype == MemType\_Normal) &&
            (attrs.inner.attrs == MemAttr\_WB) &&
            (attrs.inner.hints == MemHint\_RWA) &&
            (!attrs.inner.transient) &&
            (attrs.outer.attrs == MemAttr\_WB) &&
            (attrs.outer.hints == MemHint\_RWA) &&
            (!attrs.outer.transient));
end;
```

Library pseudocode for shared/translation/atrrs/IsWBShareable

```
// IsWBShareable()
// =====
// Determines whether the given memory attributes are iWBoWB Shareable

func IsWBShareable(memattrs : MemoryAttributes) => boolean
begin
    return (memattrs.memtype == MemType\_Normal &&
            memattrs.inner.attrs == MemAttr\_WB &&
            memattrs.outer.attrs == MemAttr\_WB &&
            memattrs.shareability IN {Shareability\_ISH, Shareability\_OSH});
end;
```

Library pseudocode for shared/translation/atrrs/NormalNCMemAttr

```
// NormalNCMemAttr()
// =====
// Normal Non-cacheable memory attributes

func NormalNCMemAttr() => MemoryAttributes
begin
    var non_cacheable : MemAttrHints;
    non_cacheable.attrs = MemAttr\_NC;

    var nc_memattrs : MemoryAttributes;
    nc_memattrs.memtype = MemType\_Normal;
    nc_memattrs.outer = non_cacheable;
    nc_memattrs.inner = non_cacheable;
    nc_memattrs.shareability = Shareability\_OSH;
    nc_memattrs.tags = MemTag\_Untagged;
    nc_memattrs.notagaccess = FALSE;

    return nc_memattrs;
end;
```

Library pseudocode for shared/translation/attrs/S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =====
// Determine whether a reserved value occupies MAIR_ELx.AttrN

func S1ConstrainUnpredictableRESMAIR(attr : bits(8), slaarch64 : boolean) => boolean
begin
  case attr of
    when '0000xx01' => return !(slaarch64 && IsFeatureImplemented(FEAT_XS));
    when '0000xxxx' => return attr[1:0] != '00';
    when '01000000' => return !(slaarch64 && IsFeatureImplemented(FEAT_XS));
    when '10100000' => return !(slaarch64 && IsFeatureImplemented(FEAT_XS));
    when '11110000' => return !(slaarch64 && IsFeatureImplemented(FEAT_MTE2));
    when 'xxxx0000' => return TRUE;
    otherwise       => return FALSE;
  end;
end;
```

Library pseudocode for shared/translation/attrs/S1DecodeMemAttrs

```
// S1DecodeMemAttrs()
// =====
// Decode MAIR-format memory attributes assigned in stage 1

func S1DecodeMemAttrs(attr_in : bits(8), sh : bits(2), slaarch64 : boolean) => MemoryAttributes
begin
    var attr : bits(8) = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, slaarch64) then
        // Map reserved encodings to an allocated encoding
        (-, attr) = ConstrainUnpredictableBits{8}(Unpredictable RESMAIR);
    end;

    var memattrs : MemoryAttributes;
    case attr of
        when '0000xxxx' => // Device memory
            memattrs.memtype = MemType Device;
            memattrs.device = DecodeDevice(attr[3:2]);
            memattrs.xs = if slaarch64 then NOT attr[0] else '1';
        when '01000000' =>
            assert slaarch64 && IsFeatureImplemented(FEAT_XS);
            memattrs.memtype = MemType Normal;
            memattrs.outer.attrs = MemAttr NC;
            memattrs.inner.attrs = MemAttr NC;
            memattrs.xs = '0';
        when '10100000' =>
            assert slaarch64 && IsFeatureImplemented(FEAT_XS);
            memattrs.memtype = MemType Normal;
            memattrs.outer.attrs = MemAttr WT;
            memattrs.outer.hints = MemHint RA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs = MemAttr WT;
            memattrs.inner.hints = MemHint RA;
            memattrs.inner.transient = FALSE;
            memattrs.xs = '0';
        when '11110000' => // Tagged memory
            assert slaarch64 && IsFeatureImplemented(FEAT_MTE2);
            memattrs.memtype = MemType Normal;
            memattrs.outer.attrs = MemAttr WB;
            memattrs.outer.hints = MemHint RWA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs = MemAttr WB;
            memattrs.inner.hints = MemHint RWA;
            memattrs.inner.transient = FALSE;
            memattrs.xs = '0';
        otherwise =>
            memattrs.memtype = MemType Normal;
            memattrs.outer = DecodeLDFAttr(attr[7:4]);
            memattrs.inner = DecodeLDFAttr(attr[3:0]);

            if (memattrs.inner.attrs == MemAttr WB &&
                memattrs.outer.attrs == MemAttr WB) then
                memattrs.xs = '0';
            else
                memattrs.xs = '1';
            end;
    end;

    memattrs.shareability = DecodeShareability(sh);
    memattrs.tags = MemTag Untagged;
    memattrs.notagaccess = FALSE;

    return memattrs;
end;

// S1DecodeMemAttrs()
// =====
// Decode MAIR-format memory attributes assigned in stage 1

func S1DecodeMemAttrs(attr_in : bits(8), sh : bits(2), slaarch64 : boolean,
    walkparams : S1TTWParams, acctype : AccessType) => MemoryAttributes
begin
    var memattrs : MemoryAttributes = S1DecodeMemAttrs(attr_in, sh, slaarch64);
    if slaarch64 && attr_in == '11110000' then
        memattrs.tags = MemTag AllocationTagged;
    elsif IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) && slaarch64 && walkparams.mtx == '1' then
        memattrs.tags = MemTag CanonicallyTagged;
    end;

    return memattrs;
end;
```

Library pseudocode for shared/translation/attrs/S2CombineS1AttrHints

```
// S2CombineS1AttrHints()
// =====
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

func S2CombineS1AttrHints(s1_attrhints : MemAttrHints, s2_attrhints : MemAttrHints) => MemAttrHints
begin
    var attrhints : MemAttrHints;

    if s1_attrhints.attrs == MemAttr\_NC || s2_attrhints.attrs == MemAttr\_NC then
        attrhints.attrs = MemAttr\_NC;
    elsif s1_attrhints.attrs == MemAttr\_WT || s2_attrhints.attrs == MemAttr\_WT then
        attrhints.attrs = MemAttr\_WT;
    else
        attrhints.attrs = MemAttr\_WB;
    end;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr\_NC then
        attrhints.hints = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;
    end;

    return attrhints;
end;
```

Library pseudocode for shared/translation/attrs/S2CombineS1Device

```
// S2CombineS1Device()
// =====
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

func S2CombineS1Device(s1_device : DeviceType, s2_device : DeviceType) => DeviceType
begin
    if s1_device == DeviceType\_nGnRnE || s2_device == DeviceType\_nGnRnE then
        return DeviceType\_nGnRnE;
    elsif s1_device == DeviceType\_nGnRE || s2_device == DeviceType\_nGnRE then
        return DeviceType\_nGnRE;
    elsif s1_device == DeviceType\_nGRE || s2_device == DeviceType\_nGRE then
        return DeviceType\_nGRE;
    else
        return DeviceType\_GRE;
    end;
end;
```


Library pseudocode for shared/translation/attrs/S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====
// Combine stage 2 with stage 1 memory attributes

func S2CombineS1MemAttrs(s1_memattrs : MemoryAttributes, s2_memattrs : MemoryAttributes,
                        s2aarch64 : boolean) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;

    if s1_memattrs.memtype == MemType\_Device && s2_memattrs.memtype == MemType\_Device then
        memattrs.memtype = MemType\_Device;
        memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elsif s1_memattrs.memtype == MemType\_Device then // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elsif s2_memattrs.memtype == MemType\_Device then // S2 Device, S1 Normal
        memattrs = s2_memattrs;
    else // S2 Normal, S1 Normal
        memattrs.memtype = MemType\_Normal;
        memattrs.inner = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);
    end;

    memattrs.tags = S2MemTagType(memattrs, s1_memattrs.tags);

    if !IsFeatureImplemented(FEAT_MTE_PERM) then
        memattrs.notagaccess = FALSE;
    else
        memattrs.notagaccess = (s2_memattrs.notagaccess &&
                                s1_memattrs.tags == MemTag\_AllocationTagged);
    end;
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);

    if (memattrs.memtype == MemType\_Normal &&
        memattrs.inner.attrs == MemAttr\_WB &&
        memattrs.outer.attrs == MemAttr\_WB) then
        memattrs.xs = '0';
    elsif s2aarch64 then
        memattrs.xs = s2_memattrs.xs AND s1_memattrs.xs;
    else
        memattrs.xs = s1_memattrs.xs;
    end;

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
end;
```

Library pseudocode for shared/translation/attrs/S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =====
// Combine stage 2 shareability with stage 1

func S2CombineS1Shareability(s1_shareability : Shareability,
                             s2_shareability : Shareability) => Shareability
begin
    if (s1_shareability == Shareability\_OSH ||
        s2_shareability == Shareability\_OSH) then
        return Shareability\_OSH;
    elsif (s1_shareability == Shareability\_ISH ||
           s2_shareability == Shareability\_ISH) then
        return Shareability\_ISH;
    else
        return Shareability\_NSH;
    end;
end;
```

Library pseudocode for shared/translation/attrs/S2DecodeCacheability

```
// S2DecodeCacheability()
// =====
// Determine the stage 2 cacheability for Normal memory

func S2DecodeCacheability(attr : bits(2)) => MemAttrHints
begin
    var s2attr : MemAttrHints;

    case attr of
        when '01' => s2attr.attrs = MemAttr\_NC; // Non-cacheable
        when '10' => s2attr.attrs = MemAttr\_WT; // Write-through
        when '11' => s2attr.attrs = MemAttr\_WB; // Write-back
        otherwise => unreachable;
    end;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints      = ARBITRARY : bits(2);
    s2attr.transient = ARBITRARY : boolean;

    return s2attr;
end;
```

Library pseudocode for shared/translation/attrs/S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// =====
// Decode stage 2 memory attributes when FWB is 0

func S2DecodeMemAttrs(attr_in : bits(4), sh : bits(2), s2aarch64 : boolean) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;

    var attr : bits(4);
    if S2ResMemAttr(s2aarch64, attr_in) then
        // Map reserved encodings to an allocated encoding
        (-, attr) = ConstrainUnpredictableBits{4}(Unpredictable\_S2RESMEMATTR);
    else
        attr = attr_in;
    end;

    case attr of
        when '00xx' => // Device memory
            memattrs.memtype      = MemType\_Device;
            memattrs.device       = DecodeDevice(attr[1:0]);
        when '0100' => // Normal, Inner+Outer WB cacheable NoTagAccess memory
            assert s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM);
            memattrs.memtype      = MemType\_Normal;
            memattrs.outer        = S2DecodeCacheability('11'); // Write-back
            memattrs.inner        = S2DecodeCacheability('11'); // Write-back
        otherwise => // Normal memory
            memattrs.memtype      = MemType\_Normal;
            memattrs.outer        = S2DecodeCacheability(attr[3:2]);
            memattrs.inner        = S2DecodeCacheability(attr[1:0]);
    end;

    memattrs.shareability = DecodeShareability(sh);

    if s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM) then
        memattrs.notagaccess = attr == '0100';
    else
        memattrs.notagaccess = FALSE;
    end;

    return memattrs;
end;
```

Library pseudocode for shared/translation/attrs/S2MemTagType

```
// S2MemTagType()
// =====
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

func S2MemTagType(s2_memattrs : MemoryAttributes, s1_tagtype : MemTagType) => MemTagType
begin
    if !IsFeatureImplemented(FEAT_MTE2) then
        return MemTag\_Untagged;
    end;

    if s1_tagtype == MemTag\_AllocationTagged && IsTaggableMemAttr(s2_memattrs) then
        return MemTag\_AllocationTagged;
    end;

    // Return what stage 1 asked for if we can, otherwise Untagged.
    if s1_tagtype != MemTag\_AllocationTagged then
        return s1_tagtype;
    end;

    return MemTag\_Untagged;
end;
```

Library pseudocode for shared/translation/attrs/S2ResMemAttr

```
// S2ResMemAttr()
// =====
// Determine whether a reserved value occupies stage 2 MemAttr field when FWB is 0

func S2ResMemAttr(s2aarch64 : boolean, attr : bits(4)) => boolean
begin
    case attr of
        when '0100' => return !(s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM));
        when '1x00' => return TRUE;
        otherwise   => return FALSE;
    end;
end;
```

Library pseudocode for shared/translation/attrs/WalkMemAttrs

```
// WalkMemAttrs()
// =====
// Retrieve memory attributes of translation table walk

func WalkMemAttrs(sh : bits(2), irgn : bits(2), orgn : bits(2)) => MemoryAttributes
begin
    var walkmemattrs : MemoryAttributes;

    walkmemattrs.memtype      = MemType\_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner       = DecodeSDFAttr(irgn);
    walkmemattrs.outer       = DecodeSDFAttr(orgn);
    walkmemattrs.tags        = MemTag\_Untagged;
    if (walkmemattrs.inner.attrs == MemAttr\_WB &&
        walkmemattrs.outer.attrs == MemAttr\_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';
    end;
    walkmemattrs.notagaccess = FALSE;

    return walkmemattrs;
end;
```

Library pseudocode for shared/translation/faults/AlignmentFault

```
// AlignmentFault()
// =====
// Return a fault record indicating an Alignment fault not due to memory type has occurred
// for a specific access

func AlignmentFault(accdesc : AccessDescriptor, vaddress : bits(64)) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, vaddress);
    fault.statuscode = Fault\_Alignment;

    return fault;
end;
```

Library pseudocode for shared/translation/faults/ExclusiveFault

```
// ExclusiveFault()
// =====
// Return a fault record indicating a fault for an unsupported Exclusive access

func ExclusiveFault(accdesc : AccessDescriptor, vaddress : bits(64)) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, vaddress);
    fault.statuscode         = Fault\_Exclusive;

    return fault;
end;
```

Library pseudocode for shared/translation/faults/NoFault

```
// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred

func NoFault() => FaultRecord
begin
    var fault : FaultRecord;

    fault.vaddress = ARBITRARY : bits(64);
    fault.statuscode = Fault\_None;
    fault.accessdesc = ARBITRARY : AccessDescriptor;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
    fault.dirtybit = FALSE;
    fault.overlay = FALSE;
    fault.toplevel = FALSE;
    fault.assuredonly = FALSE;
    fault.sltagnodata = FALSE;
    fault.tagaccess = FALSE;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();
    fault.hdbssf = FALSE;

    return fault;
end;

// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred for a specific access

func NoFault(accdesc : AccessDescriptor) => FaultRecord
begin
    var fault : FaultRecord;

    fault.statuscode = Fault\_None;
    fault.accessdesc = accdesc;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
    fault.dirtybit = FALSE;
    fault.overlay = FALSE;
    fault.toplevel = FALSE;
    fault.assuredonly = FALSE;
    fault.sltagnodata = FALSE;
    fault.tagaccess = FALSE;
    fault.write = !accdesc.read && accdesc.write;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();
    fault.hdbssf = FALSE;

    return fault;
end;

// NoFault()
// =====

func NoFault(accdesc : AccessDescriptor, vaddress : bits(64)) => FaultRecord
begin
    var fault : FaultRecord = NoFault();
    fault.accessdesc = accdesc;
    fault.write = !accdesc.read && accdesc.write;
    fault.vaddress = vaddress;

    return fault;
end;
```

Library pseudocode for shared/translation/gpc/AbovePPS

```
// AbovePPS()
// =====
// Returns TRUE if an address exceeds the range configured in GPCCR_EL3.PPS.

func AbovePPS(address : bits(NUM_PABITS)) => boolean
begin
    let pps : integer{} = DecodePPS();
    if pps >= NUM_PABITS then
        return FALSE;
    end;

    return !IsZero(address[NUM_PABITS-1:pps]);
end;
```

Library pseudocode for shared/translation/gpc/DecodeGPTBlock

```
// DecodeGPTBlock()
// =====
// Decode a GPT Block descriptor.

func DecodeGPTBlock(pgs : PGSe, gpt_entry : bits(64)) => GPTEntry
begin
    assert gpt_entry[3:0] == GPT_Block;
    var result : GPTEntry;
    result.gpi = gpt_entry[7:4];
    result.level = 0;
    result.istable = FALSE;

    // GPT information from a level 0 GPT Block descriptor is permitted
    // to be cached in a TLB as though the Block is a contiguous region
    // of granules each of the size configured in GPCCR_EL3.PGS.
    case pgs of
        when PGS_4KB => result.size = GPTRange_4KB;
        when PGS_16KB => result.size = GPTRange_16KB;
        when PGS_64KB => result.size = GPTRange_64KB;
        otherwise => unreachable;
    end;
    result.contig_size = GPTL0Size();

    return result;
end;
```

Library pseudocode for shared/translation/gpc/DecodeGPTContiguous

```
// DecodeGPTContiguous()
// =====
// Decode a GPT Contiguous descriptor.

func DecodeGPTContiguous(pgs : PGSe, gpt_entry : bits(64)) => GPTEntry
begin
    assert gpt_entry[3:0] == GPT_Contig;
    var result : GPTEntry;
    result.gpi = gpt_entry[7:4];

    case pgs of
        when PGS_4KB => result.size = GPTRange_4KB;
        when PGS_16KB => result.size = GPTRange_16KB;
        when PGS_64KB => result.size = GPTRange_64KB;
        otherwise => unreachable;
    end;

    case gpt_entry[9:8] of
        when '01' => result.contig_size = GPTRange_2MB;
        when '10' => result.contig_size = GPTRange_32MB;
        when '11' => result.contig_size = GPTRange_512MB;
        otherwise => unreachable;
    end;

    result.level = 1;
    result.istable = FALSE;

    return result;
end;
```

Library pseudocode for shared/translation/gpc/DecodeGPTGranules

```
// DecodeGPTGranules()
// =====
// Decode a GPT Granules descriptor.

func DecodeGPTGranules(pgs : PGSe, index : integer, gpt_entry : bits(64)) => GPTEntry
begin
    var result : GPTEntry;
    result.gpi = gpt_entry[index*4 +:4];

    case pgs of
        when PGS_4KB => result.size = GPTRange_4KB;
        when PGS_16KB => result.size = GPTRange_16KB;
        when PGS_64KB => result.size = GPTRange_64KB;
        otherwise => unreachable;
    end;

    result.contig_size = result.size; // No contiguity
    result.level = 1;
    result.istable = FALSE;

    return result;
end;
```

Library pseudocode for shared/translation/gpc/DecodeGPTTable

```
// DecodeGPTTable()
// =====
// Decode a GPT Table descriptor.

func DecodeGPTTable(pgs : PGSe, gpt_entry : bits(64)) => GPTTable
begin
    assert gpt_entry[3:0] == GPT_Table;
    var result : GPTTable;

    // Descriptor bits [LOGPTSZ-PGS-2:12] are RES0
    var s : integer{30, 34, 36, 39};
    case GPTLOSize() of
        when GPTRange_1GB => s = 30;
        when GPTRange_16GB => s = 34;
        when GPTRange_64GB => s = 36;
        when GPTRange_512GB => s = 39;
        otherwise => unreachable;
    end;

    var p : integer{12, 14, 16};
    case pgs of
        when PGS_4KB => p = 12;
        when PGS_16KB => p = 14;
        when PGS_64KB => p = 16;
        otherwise => unreachable;
    end;

    let align_bits : integer{} = (s - p) - 1;
    result.address = AlignDownP2(gpt_entry[NUM_PABITS-1:0], align_bits);

    return result;
end;
```

Library pseudocode for shared/translation/gpc/DecodePGS

```
// DecodePGS()
// =====

func DecodePGS(pgs : bits(2)) => PGSe
begin
    case pgs of
        when '00' => return PGS_4KB;
        when '10' => return PGS_16KB;
        when '01' => return PGS_64KB;
        otherwise => unreachable;
    end;
end;
```

Library pseudocode for shared/translation/gpc/DecodePGSRange

```
// DecodePGSRange()
// =====

func DecodePGSRange(pgs : PGS_e) => AddressSize
begin
  case pgs of
    when PGS_4KB => return GPtrange_4KB;
    when PGS_16KB => return GPtrange_16KB;
    when PGS_64KB => return GPtrange_64KB;
    otherwise => unreachable;
  end;
end;
```

Library pseudocode for shared/translation/gpc/DecodePPS

```
// DecodePPS()
// =====
// Size of region protected by the GPT, in bits.

func DecodePPS() => AddressSize
begin
  case GPCCR_EL3().[PPS3, PPS] of
    when '0000' => return 32;
    when '0001' => return 36;
    when '0010' => return 40;
    when '0011' => return 42;
    when '0100' => return 44;
    when '0101' => return 48;
    when '0110' => return 52;
    when '0111' => assert IsFeatureImplemented(FEAT_RME_GPC3); return 56;
    when '1000' => assert IsFeatureImplemented(FEAT_RME_GPC3); return 46;
    when '1001' => assert IsFeatureImplemented(FEAT_RME_GPC3); return 47;
    otherwise => unreachable;
  end;
end;
```

Library pseudocode for shared/translation/gpc/GPCBW_EL3BWSTRIDEValid

```
// GPCBW_EL3BWSTRIDEValid()
// =====
// Returns whether the current GPCBW_EL3.BWSTRIDE value is valid

func GPCBW_EL3BWSTRIDEValid() => boolean
begin
  assert IsFeatureImplemented(FEAT_RME_GPC3);
  return GPCBW_EL3().BWSTRIDE IN {
    '00000',
    '00010',
    '00100',
    '00110',
    '00111',
    '01000',
    '01001',
    '01010',
    '10000'
  };
end;
```

Library pseudocode for shared/translation/gpc/GPCFault

```
// GPCFault()
// =====
// Constructs and returns a GPCF

func GPCFault(gpf : GPCF, level : integer) => GPCFRecord
begin
  var fault : GPCFRecord;
  fault.gpf = gpf;
  fault.level = level;
  return fault;
end;
```

Library pseudocode for shared/translation/gpc/GPCNoFault

```
// GPCNoFault()
// =====
// Returns the default properties of a GPCF that does not represent a fault

func GPCNoFault() => GPCFRecord
begin
    var result : GPCFRecord;
    result.gpf = GPCF\_None;
    return result;
end;
```

Library pseudocode for shared/translation/gpc/GPCRegistersConsistent

```
// GPCRegistersConsistent()
// =====
// Returns whether the GPT registers are configured correctly.
// This returns false if any fields select a Reserved value.

func GPCRegistersConsistent() => boolean
begin
    // Check for Reserved register values
    if IsFeatureImplemented(FEAT_RME_GPC3) then
        if ! GPCCR_EL3().[PPS3, PPS] IN {'0xxx', '100x'} then
            return FALSE;
        end;

        if GPCCR_EL3().GPCBW == '1' then
            if ! GPCBW_EL3().BWSIZE IN {'00x', '1x0', '010'} then
                return FALSE;
            end;

            if !GPCBW\_EL3BWSTRIDEValid() then
                return FALSE;
            end;

            let bwstride : integer = 1 << (40 + UInt(GPCBW_EL3().BWSTRIDE));
            let bwaddr : integer = UInt(GPCBW_EL3().BWADDR) << 30;
            if bwstride <= bwaddr then
                return FALSE;
            end;

            if !IsAlignedSize{26}(GPCBW_EL3().BWADDR, 1 << UInt(GPCBW_EL3().BWSIZE)) then
                return FALSE;
            end;
        end;
    else
        if GPCCR_EL3().PPS == '111' then
            return FALSE;
        end;
    end;

    if DecodePPS() > AArch64\_PAMax() then
        return FALSE;
    end;
    if GPCCR_EL3().PGS == '11' then
        return FALSE;
    end;
    if GPCCR_EL3().SH == '01' then
        return FALSE;
    end;

    // Inner and Outer Non-cacheable requires Outer Shareable
    if GPCCR_EL3().[ORGN, IRGN] == '0000' && GPCCR_EL3().SH != '10' then
        return FALSE;
    end;

    return TRUE;
end;
```


Library pseudocode for shared/translation/gpc/GPICheck

```
// GPICheck()
// =====
// Returns whether an access to a given physical address space is permitted
// given the configured GPI value.
// paspace: Physical address space of the access
// gpi: Value read from GPT for the access
// ss: Security state of the access

func GPICheck(paspace : PASpace, gpi : bits(4), ss : SecurityState) => boolean
begin
  case gpi of
    when GPT\_NoAccess =>
      return FALSE;
    when GPT\_Secure =>
      assert IsFeatureImplemented(FEAT_SEL2);
      return paspace == PAS\_Secure;
    when GPT\_NonSecure =>
      return paspace == PAS\_NonSecure;
    when GPT\_Root =>
      return paspace == PAS\_Root;
    when GPT\_Realm =>
      return paspace == PAS\_Realm;
    when GPT\_NonSecureOnly =>
      assert IsFeatureImplemented(FEAT_RME_GPC2);
      return paspace == PAS\_NonSecure && (ss IN {SS\_Root, SS\_NonSecure});
    when GPT\_SystemAgent =>
      assert IsFeatureImplemented(FEAT_RME_GDI);
      return paspace == PAS\_SystemAgent;
    when GPT\_NonSecureProtected =>
      assert IsFeatureImplemented(FEAT_RME_GDI);
      return paspace == PAS\_NonSecureProtected;
    when GPT\_NA6 =>
      assert IsFeatureImplemented(FEAT_RME_GDI);
      return FALSE;
    when GPT\_NA7 =>
      assert IsFeatureImplemented(FEAT_RME_GDI);
      return FALSE;
    when GPT\_Any =>
      return TRUE;
    otherwise =>
      unreachable;
  end;
end;
```

Library pseudocode for shared/translation/gpc/GPIIndex

```
// GPIIndex()
// =====

func GPIIndex(pa : bits(NUM\_PABITS)) => integer
begin
  case DecodePGS(GPCCR_EL3().PGS) of
    when PGS\_4KB => return UInt(pa[15:12]);
    when PGS\_16KB => return UInt(pa[17:14]);
    when PGS\_64KB => return UInt(pa[19:16]);
    otherwise => unreachable;
  end;
end;
```

Library pseudocode for shared/translation/gpc/GPIValid

```
// GPIValid()
// =====
// Returns whether a given value is a valid encoding for a GPI value

func GPIValid(gpi : bits(4)) => boolean
begin
  case gpi of
    when GPT\_NoAccess =>
      return TRUE;
    when GPT\_NonSecureProtected =>
      return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3().NSP == '1';
    when GPT\_SystemAgent =>
      return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3().SA == '1';
    when GPT\_NA6 =>
      return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3().NA6 == '1';
    when GPT\_NA7 =>
      return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3().NA7 == '1';
    when GPT\_Secure =>
      return IsFeatureImplemented(FEAT_SEL2);
    when GPT\_NonSecure =>
      return TRUE;
    when GPT\_Realm =>
      return TRUE;
    when GPT\_Root =>
      return TRUE;
    when GPT\_NonSecureOnly =>
      return IsFeatureImplemented(FEAT_RME_GPC2) && GPCCR_EL3().NSO == '1';
    when GPT\_Any =>
      return TRUE;
    otherwise =>
      return FALSE;
  end;
end;
```

Library pseudocode for shared/translation/gpc/GPT

```
// GPT dimensions
// =====

constant GPTRange_4KB : AddressSize = 12;
constant GPTRange_16KB : AddressSize = 14;
constant GPTRange_64KB : AddressSize = 16;
constant GPTRange_2MB : AddressSize = 21;
constant GPTRange_32MB : AddressSize = 25;
constant GPTRange_512MB : AddressSize = 29;
constant GPTRange_1GB : AddressSize = 30;
constant GPTRange_16GB : AddressSize = 34;
constant GPTRange_64GB : AddressSize = 36;
constant GPTRange_512GB : AddressSize = 39;
```

Library pseudocode for shared/translation/gpc/GPTBlockDescriptorValid

```
// GPTBlockDescriptorValid()
// =====
// Returns TRUE if the given GPT Block descriptor is valid, and FALSE otherwise.

func GPTBlockDescriptorValid(level_0_entry : bits(64)) => boolean
begin
  assert level_0_entry[3:0] == GPT\_Block;
  return IsZero(level_0_entry[63:8]) && GPIValid(level_0_entry[7:4]);
end;
```

Library pseudocode for shared/translation/gpc/GPTContigDescriptorValid

```
// GPTContigDescriptorValid()
// =====
// Returns TRUE if the given GPT Contiguous descriptor is valid, and FALSE otherwise.

func GPTContigDescriptorValid(level_1_entry : bits(64)) => boolean
begin
  assert level_1_entry[3:0] == GPT\_Contig;
  return (IsZero(level_1_entry[63:10]) &&
    !IsZero(level_1_entry[9:8]) &&
    GPIValid(level_1_entry[7:4]));
end;
```

Library pseudocode for shared/translation/gpc/GPTEntry

```
// GPTEntry
// =====

type GPTEntry of record {
    gpi      : bits(4),           // GPI value for this region
    size     : AddressSize,      // Region size
    contig_size : AddressSize,    // Contiguous region size
    istable   : boolean,         // Flag a table entry
    level     : integer,         // Level of GPT lookup
    pa       : bits(NUM\_PABITS) // PA uniquely identifying the GPT entry
};
```

Library pseudocode for shared/translation/gpc/GPTGranulesDescriptorValid

```
// GPTGranulesDescriptorValid()
// =====
// Returns TRUE if the given GPT Granules descriptor is valid, and FALSE otherwise.

func GPTGranulesDescriptorValid(level_1_entry : bits(64)) => boolean
begin
    for i = 0 to 15 do
        if !GPIValid(level_1_entry[i*4 +:4]) then
            return FALSE;
        end;
    end;
    return TRUE;
end;
```

Library pseudocode for shared/translation/gpc/GPTL0Size

```
// GPTL0Size()
// =====
// Returns number of bits covered by a level 0 GPT entry

func GPTL0Size() => AddressSize
begin
    case GPCCR_EL3().LOGPSSZ of
        when '0000' => return GPTRange\_1GB;
        when '0100' => return GPTRange\_16GB;
        when '0110' => return GPTRange\_64GB;
        when '1001' => return GPTRange\_512GB;
        otherwise => unreachable;
    end;
    return 30;
end;
```

Library pseudocode for shared/translation/gpc/GPTLevel0EntryValid

```
// GPTLevel0EntryValid()
// =====
// Returns TRUE if the given level 0 gpt descriptor is valid, and FALSE otherwise.

func GPTLevel0EntryValid(gpt_entry : bits(64)) => boolean
begin
    case gpt_entry[3:0] of
        when GPT\_Block => return GPTBlockDescriptorValid(gpt_entry);
        when GPT\_Table => return GPTTableDescriptorValid(gpt_entry);
        otherwise => return FALSE;
    end;
end;
```

Library pseudocode for shared/translation/gpc/GPTLevel0Index

```
// GPTLevel0Index()
// =====
// Compute the level 0 index based on input PA.

func GPTLevel0Index(pa : bits(NUM\_PABITS)) => integer
begin
    // Input address and index bounds
    let pps : integer{} = DecodePPS();
    let l0sz : integer{} = GPTL0Size();
    if pps <= l0sz then
        return 0;
    end;

    return UInt(pa[pps-1:l0sz]);
end;
```

Library pseudocode for shared/translation/gpc/GPTLevel1EntryValid

```
// GPTLevel1EntryValid()
// =====
// Returns TRUE if the given level 1 gpt descriptor is valid, and FALSE otherwise.

func GPTLevel1EntryValid(gpt_entry : bits(64)) => boolean
begin
  case gpt_entry[3:0] of
    when GPT Contig => return GPTContigDescriptorValid(gpt_entry);
    otherwise =>      return GPTGranulesDescriptorValid(gpt_entry);
  end;
end;
```

Library pseudocode for shared/translation/gpc/GPTLevel1Index

```
// GPTLevel1Index()
// =====
// Compute the level 1 index based on input PA.

func GPTLevel1Index(pa : bits(NUM\_PABITS)) => integer
begin
  // Input address and index bounds
  let l0sz : integer{} = GPTL0Size();
  case DecodePGS(GPCCR_EL3().PGS) of
    when PGS 4KB => return UInt(pa[l0sz-1:16]);
    when PGS 16KB => return UInt(pa[l0sz-1:18]);
    when PGS 64KB => return UInt(pa[l0sz-1:20]);
    otherwise => unreachable;
  end;
end;
```

Library pseudocode for shared/translation/gpc/GPTTable

```
// GPTTable
// =====

type GPTTable of record {
  address : bits(NUM\_PABITS) // Base address of next table
};
```

Library pseudocode for shared/translation/gpc/GPTTableDescriptorValid

```
// GPTTableDescriptorValid()
// =====
// Returns TRUE if the given GPT Table descriptor is valid, and FALSE otherwise.

func GPTTableDescriptorValid(level_0_entry : bits(64)) => boolean
begin
  assert level_0_entry[3:0] == GPT Table;
  let l0sz : integer{} = GPTL0Size();
  let pgs : PGSe = DecodePGS(GPCCR_EL3().PGS);
  let p : integer{} = DecodePGSRange(pgs);
  let top : integer{} = if DecodePPS() == 56 then 56 else 52;
  return IsZero(level_0_entry[63:top,11:4]) && IsZero(level_0_entry[(l0sz-p)-2:12]);
end;
```



```

// GPTWalk()
// =====
// Get the GPT entry for a given physical address, pa

func GPTWalk(pa : bits(NUM_PABITS), accdesc : AccessDescriptor) => (GPCFRecord, GPTEntry)
begin
    // GPT base address
    var base : bits(NUM_PABITS);
    let pgs : PGSe = DecodePGS(GPCCR_EL3().PGS);

    // The level 0 GPT base address is aligned to the greater of:
    // * the size of the level 0 GPT, determined by GPCCR_EL3().[PPS, LOGPTSZ].
    // * 4KB
    base = ZeroExtend(NUM_PABITS)(GPTBR_EL3().BADDR::Zeros{12});
    let pps : AddressSize = DecodePPS();
    let l0sz : integer{} = GPTL0Size();
    let alignment : integer{} = Max((pps - l0sz) + 3, 12) as AddressSize;
    base[alignment-1:0] = Zeros{alignment};

    let gptaccdesc : AccessDescriptor = CreateAccDescGPTW(accdesc);

    // Access attributes and address for GPT fetches
    var gptaddrdesc : AccessDescriptor;
    gptaddrdesc.memattrs = WalkMemAttrs(GPCCR_EL3().SH, GPCCR_EL3().IRGN, GPCCR_EL3().ORGN);
    gptaddrdesc.fault = NoFault(gptaccdesc);

    gptaddrdesc.paddress.paspace = PAS_Root;
    gptaddrdesc.paddress.address = base + GPTLevel0Index(pa) * 8;

    // Fetch LOGPT entry
    var level_0_entry : bits(64);
    var memstatus : PhysMemRetStatus;
    (memstatus, level_0_entry) = PhysMemRead{64}(gptaddrdesc, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 0), ARBITRARY : GPTEntry);
    end;

    if !GPTLevel0EntryValid(level_0_entry) then
        return (GPCFault(GPCF_Walk, 0), ARBITRARY : GPTEntry);
    end;

    var result : GPTEntry;
    var table : GPTTable;
    case level_0_entry[3:0] of
        when GPT_Block =>
            // Decode the GPI value and return that
            result = DecodeGPTBlock(pgs, level_0_entry);
            result.pa = pa;
            return (GPCNoFault(), result);
        when GPT_Table =>
            // Decode the table entry and continue walking
            table = DecodeGPTTable(pgs, level_0_entry);
            // The address must be within the range covered by the GPT
            if AbovePPS(table.address) then
                return (GPCFault(GPCF_AddressSize, 0), ARBITRARY : GPTEntry);
            end;
        otherwise =>
            // An invalid encoding would be caught by GPTLevel0EntryValid()
            unreachable;
    end;

    // Must be a GPT Table entry
    assert level_0_entry[3:0] == GPT_Table;

    // Address of level 1 GPT entry
    let offset : integer = GPTLevel1Index(pa) * 8;

    var level_1_entry : bits(64);

    if IsFeatureImplemented(FEAT_RME_GDI) then
        // When FEAT_RME_GDI is implemented, the descriptor validation checks are performed
        // on a pair of descriptors within a naturally aligned 16-byte region of memory.
        gptaddrdesc.paddress.address = AlignDownSize{56}(table.address + offset, 16);
        var level_1_entry_lower : bits(64);
        (memstatus, level_1_entry_lower) = PhysMemRead{64}(gptaddrdesc, gptaccdesc);
        if IsFault(memstatus) then
            return (GPCFault(GPCF_EABT, 1), ARBITRARY : GPTEntry);
        end;

        gptaddrdesc.paddress.address = gptaddrdesc.paddress.address + 8;
        var level_1_entry_upper : bits(64);
        (memstatus, level_1_entry_upper) = PhysMemRead{64}(gptaddrdesc, gptaccdesc);
        if IsFault(memstatus) then
            return (GPCFault(GPCF_EABT, 1), ARBITRARY : GPTEntry);
        end;

        // An individual GPT descriptor is valid when both descriptors within the pair are valid.
        if (!GPTLevel1EntryValid(level_1_entry_upper) ||
            !GPTLevel1EntryValid(level_1_entry_lower)) then

```

```

        return (GPCFault(GPCF_Walk, 1), ARBITRARY : GPTEntry);
    end;

    if offset[3] == '1' then
        level_1_entry = level_1_entry_upper;
    else
        level_1_entry = level_1_entry_lower;
    end;
else
    gptaddrdesc.paddress.address = table.address + offset;
    // Fetch L1GPT entry
    (memstatus, level_1_entry) = PhysMemRead{64}(gptaddrdesc, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 1), ARBITRARY : GPTEntry);
    end;

    if !GPTLevel1EntryValid(level_1_entry) then
        return (GPCFault(GPCF_Walk, 1), ARBITRARY : GPTEntry);
    end;
end;

case level_1_entry[3:0] of
    when GPT_Contig =>
        result = DecodeGPTContiguous(pgs, level_1_entry);
    otherwise =>
        let gpi_index : integer = GPIIndex(pa);
        result = DecodeGPTGranules(pgs, gpi_index, level_1_entry);
end;

result.pa = pa;
return (GPCNoFault(), result);
end;

```

Library pseudocode for shared/translation/gpc/GranuleProtectionCheck

```
// GranuleProtectionCheck()
// =====
// Returns whether a given access is permitted, according to the
// granule protection check.
// addrdesc and accdesc describe the access to be checked.

func GranuleProtectionCheck(addrdesc : AddressDescriptor, accdesc : AccessDescriptor) => GPCFRecord
begin

    assert IsFeatureImplemented(FEAT_RME);
    // The address to be checked
    let address = addrdesc.paddress;

    // Bypass mode - all accesses pass
    if GPCCR_EL3().GPC == '0' then
        return GPCNoFault();
    end;

    // Configuration consistency check
    if !GPCRegistersConsistent() then
        return GPCFault(GPCF\_Walk, 0);
    end;

    if IsFeatureImplemented(FEAT_RME_GPC2) then
        var access_disabled : boolean;

        case address.paspace of
            when PAS\_Secure => access_disabled = GPCCR_EL3().SPAD == '1';
            when PAS\_NonSecure => access_disabled = GPCCR_EL3().NSPAD == '1';
            when PAS\_Realm => access_disabled = GPCCR_EL3().RLPAD == '1';
            when PAS\_Root => access_disabled = FALSE;
            otherwise => unreachable;
        end;

        if access_disabled then
            return GPCFault(GPCF\_Fail, 0);
        end;
    end;

    // Input address size check
    if AbovePPS(address.address) then
        if (address.paspace == PAS\_NonSecure ||
            (IsFeatureImplemented(FEAT_RME_GPC2) && GPCCR_EL3().APPSAA == '1')) then
            return GPCNoFault();
        else
            return GPCFault(GPCF\_Fail, 0);
        end;
    end;

    if (IsFeatureImplemented(FEAT_RME_GPC3) && GPCCR_EL3().GPCBW == '1' &&
        PAWithinGPGBypassWindow(address.address)) then
        return GPCNoFault();
    end;

    // GPT base address size check
    let gpt_base : bits(NUM\_PABITS) = ZeroExtend{} (GPTBR_EL3().BADDR::Zeros{12});
    if AbovePPS(gpt_base) then
        return GPCFault(GPCF\_AddressSize, 0);
    end;

    // GPT lookup
    var (gpcf, gpt_entry) = GPTWalk(address.address, accdesc);
    if gpcf.gpf != GPCF\_None then
        return gpcf;
    end;

    // Check input physical address space against GPI
    let permitted : boolean = GPICheck(address.paspace, gpt_entry.gpi, accdesc.ss);

    if !permitted then
        gpcf = GPCFault(GPCF\_Fail, gpt_entry.level);
        return gpcf;
    end;

    // Check passed

    return GPCNoFault();
end;
```


Library pseudocode for shared/translation/gpc/IsGranuleProtectionCheckedAccess

```
// IsGranuleProtectionCheckedAccess()
// =====
// Check if the access should be subject to Granule protection check returns
// true if it is, false otherwise

func IsGranuleProtectionCheckedAccess(accdesc : AccessDescriptor) => boolean
begin
    if accdesc.acctype == AccessType\_DC then
        return ImpDefBool("GPC Fault on DC operations");
    end;

    return TRUE;
end;
```

Library pseudocode for shared/translation/gpc/PAWithinGPCCBypassWindow

```
// PAWithinGPCCBypassWindow()
// =====
// Check if the supplied address is within a GPC Bypass window.

func PAWithinGPCCBypassWindow(pa_in : bits(56)) => boolean
begin
    // Only check the top 26 bits as the minimum window size is 1GB
    let pa : bits(26) = pa_in[55:30];

    let gpcbw1 : integer{} = UInt(GPCBW_EL3().BWSIZE);
    let gpcbwu : integer{} = 9 + UInt(GPCBW_EL3().BWSTRIDE);

    return pa[gpcbwu:gpcbw1] == GPCBW_EL3().BWADDR[gpcbwu:gpcbw1];
end;
```

Library pseudocode for shared/translation/gpc/PGS

```
// PGS
// ===
// Physical granule size

type PGSe of enumeration {
    PGS_4KB,
    PGS_16KB,
    PGS_64KB
};
```

Library pseudocode for shared/translation/gpc/Table

```
// Table format information
// =====
// Granule Protection Table constants

constant GPT_NoAccess : bits(4)      = '0000';
constant GPT_Table : bits(4)         = '0011';
constant GPT_Block : bits(4)         = '0001';
constant GPT_Contig : bits(4)        = '0001';
constant GPT_SystemAgent : bits(4)   = '0100';
constant GPT_NonSecureProtected : bits(4) = '0101';
constant GPT_NA6 : bits(4)           = '0110';
constant GPT_NA7 : bits(4)           = '0111';
constant GPT_Secure : bits(4)         = '1000';
constant GPT_NonSecure : bits(4)      = '1001';
constant GPT_Root : bits(4)           = '1010';
constant GPT_Realm : bits(4)          = '1011';
constant GPT_NonSecureOnly : bits(4)  = '1101';
constant GPT_Any : bits(4)            = '1111';
```

Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====
// Stage 1 translation regime for the given Exception level

readonly func S1TranslationRegime(el : bits(2)) => bits(2)
begin
    if el != EL0 then
        return el;

    elsif HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && SCR().NS == '0' then
        return EL3;
    elsif IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
        return EL2;
    else
        return EL1;
    end;
end;

// S1TranslationRegime()
// =====
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the System register accessors (SCTLR_ELx, etc.) implicitly
// return the correct value.

readonly func S1TranslationRegime() => bits(2)
begin
    return S1TranslationRegime\(PSTATE.EL\);
end;
```

Library pseudocode for shared/translation/vmsa/AddressDescriptor

```
// AddressDescriptor
// =====
// Descriptor used to access the underlying memory array.

type AddressDescriptor of record {
    fault : FaultRecord, // fault.statuscode indicates whether the address is valid
    memattrs : MemoryAttributes,
    paddress : FullAddress,
    s1assured : boolean, // Stage 1 Assured Translation Property
    s2fslmro : boolean, // Stage 2 MRO permission for Stage 1
    mecid : bits(16), // FEAT_MEC: Memory Encryption Context ID
    vaddress : bits(64)
};

constant FINAL_LEVEL : integer = 3;
```

Library pseudocode for shared/translation/vmsa/ContiguousSize

```
// ContiguousSize()
// =====
// Return the number of entries log 2 marking a contiguous output range

func ContiguousSize(dl28 : bit, tgx : TGx, level : integer) => integer
begin
    if dl28 == '1' then
        case tgx of
            when TGx_4KB =>
                assert level IN {1, 2, 3};
                return if level == 1 then 2 else 4;
            when TGx_16KB =>
                assert level IN {1, 2, 3};
                if level == 1 then
                    return 2;
                elsif level == 2 then
                    return 4;
                else
                    return 6;
                end;
            when TGx_64KB =>
                assert level IN {2, 3};
                return if level == 2 then 6 else 4;
        end;
    else
        case tgx of
            when TGx_4KB =>
                assert level IN {1, 2, 3};
                return 4;
            when TGx_16KB =>
                assert level IN {2, 3};
                return if level == 2 then 5 else 7;
            when TGx_64KB =>
                assert level IN {2, 3};
                return 5;
        end;
    end;
end;
```

Library pseudocode for shared/translation/vmsa/CreateAddressDescriptor

```
// CreateAddressDescriptor()
// =====
// Set internal members for address descriptor type to valid values

func CreateAddressDescriptor(pa : FullAddress, memattrs : MemoryAttributes,
                             accdesc : AccessDescriptor) => AddressDescriptor
begin
    var addrdesc : AddressDescriptor;

    addrdesc.paddress = pa;
    addrdesc.memattrs = memattrs;
    addrdesc.fault     = NoFault(accdesc);

    return addrdesc;
end;

// CreateAddressDescriptor()
// =====
// Set internal members for address descriptor type to valid values

func CreateAddressDescriptor(va : bits(64), pa : FullAddress,
                             memattrs : MemoryAttributes,
                             accdesc : AccessDescriptor) => AddressDescriptor
begin
    var addrdesc : AddressDescriptor;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault     = NoFault(accdesc);
    addrdesc.slassured = FALSE;

    return addrdesc;
end;
```

Library pseudocode for shared/translation/vmsa/CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// =====
// Set internal members for address descriptor type with values indicating error

func CreateFaultyAddressDescriptor(fault : FaultRecord) => AddressDescriptor
begin
    var addrdesc : AddressDescriptor;

    addrdesc.vaddress = fault.vaddress;
    addrdesc.fault     = fault;

    return addrdesc;
end;
```

Library pseudocode for shared/translation/vmsa/DecodePASpace

```
// DecodePASpace()
// =====
// Decode the target PA Space

func DecodePASpace(nse2 : bit, nse : bit, ns : bit) => PASpace
begin
    case nse2::nse::ns of
        when '000' => return PAS\_Secure;
        when '001' => return PAS\_NonSecure;
        when '010' => return PAS\_Root;
        when '011' => return PAS\_Realm;
        when '100' => return PAS\_SystemAgent;
        when '101' => return PAS\_NonSecureProtected;
        when '110' => return PAS\_NA6;
        when '111' => return PAS\_NA7;
        otherwise => unreachable;
    end;
end;
```

Library pseudocode for shared/translation/vmsa/DescriptorType

```
// DescriptorType
// =====
// Translation table descriptor formats

type DescriptorType of enumeration {
    DescriptorType_Table,
    DescriptorType_Leaf,
    DescriptorType_Invalid
};
```

Library pseudocode for shared/translation/vmsa/Domains

```
// Domains
// =====
// Short-descriptor format Domains

constant Domain_NoAccess : bits(2) = '00';
constant Domain_Client   : bits(2) = '01';
constant Domain_Manager  : bits(2) = '11';
```

Library pseudocode for shared/translation/vmsa/FetchDescriptor

```
// FetchDescriptor()
// =====
// Fetch a translation table descriptor

func FetchDescriptor(N : integer{32, 64, 128})(ee : bit, walkaddress : AddressDescriptor,
                                                walkaccess : AccessDescriptor,
                                                fault_in : FaultRecord) => (FaultRecord, bits(N))

begin
    var descriptor : bits(N);
    var fault : FaultRecord = fault_in;

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(walkaddress, walkaccess);
        if fault.gpcf.gpf != GPCF\_None then
            fault.statuscode = Fault\_GPCFOnWalk;
            fault.paddress = walkaddress.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, ARBITRARY : bits(N));
        end;
    end;

    var memstatus : PhysMemRetStatus;
    (memstatus, descriptor) = PhysMemRead{N}(walkaddress, walkaccess);
    if IsFault(memstatus) then
        let iswrite : boolean = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, walkaddress,
                                         walkaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, ARBITRARY : bits(N));
        end;
    end;

    if ee == '1' then
        descriptor = BigEndianReverse{N}(descriptor);
    end;

    return (fault, descriptor);
end;
```

Library pseudocode for shared/translation/vmsa/HasUnprivileged

```
// HasUnprivileged()
// =====
// Returns whether a translation regime serves EL0 as well as a higher EL

readonly func HasUnprivileged(regime : Regime) => boolean

begin
    return (regime IN {
        Regime\_EL20,
        Regime\_EL30,
        Regime\_EL10
    });
end;
```

Library pseudocode for shared/translation/vmsa/NUM_PABITS

```
// NUM_PABITS
// =====

constant NUM_PABITS : integer{} = 56;
```

Library pseudocode for shared/translation/vmsa/Regime

```
// Regime
// =====
// Translation regimes

type Regime of enumeration {
    Regime_EL3,           // EL3
    Regime_EL30,          // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2,           // EL2
    Regime_EL20,          // EL2&0
    Regime_EL10           // EL1&0
};
```

Library pseudocode for shared/translation/vmsa/RegimeUsingAArch32

```
// RegimeUsingAArch32()
// =====
// Determine if the EL controlling the regime executes in AArch32 state

func RegimeUsingAArch32(regime : Regime) => boolean
begin
    case regime of
        when Regime\_EL10 => return ELUsingAArch32\(EL1\);
        when Regime\_EL30 => return TRUE;
        when Regime\_EL20 => return FALSE;
        when Regime\_EL2   => return ELUsingAArch32\(EL2\);
        when Regime\_EL3   => return FALSE;
    end;
end;
```

Library pseudocode for shared/translation/vmsa/S1TTWParams

```
// S1TTWParams
// =====
// Register fields corresponding to stage 1 translation
// For A32-VMSA, if noted, they correspond to A32-LPAE (Long descriptor format)

type S1TTWParams of record {
// A64-VMSA exclusive parameters
    ha : bit,           // TCR_ELx.HA
    hd : bit,           // TCR_ELx.HD
    tbi : bit,          // TCR_ELx.TBI{x}
    tbid : bit,         // TCR_ELx.TBID{x}
    nfd : bit,          // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    e0pd : bit,         // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    d128 : bit,         // TCR_ELx.D128
    aie : bit,          // (TCR2_ELx/TCR_EL3).AIE
    mair2 : MAIRType,   // MAIR2_ELx
    ds : bit,           // TCR_ELx.DS
    ps : bits(3),       // TCR_ELx.{I}PS
    txsz : bits(6),     // TCR_ELx.TxSZ
    epan : bit,         // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    dct : bit,          // HCR_EL2.DCT
    nv1 : bit,          // HCR_EL2.NV1
    cmow : bit,         // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'
    pnch : bit,         // TCR{2}_ELx.PnCH
    disch : bit,        // TCR{2}_ELx.DisCH
    haft : bit,         // TCR{2}_ELx.HAFT
    mtX : bit,          // TCR_ELx.MTX{y}
    skl : bits(2),      // TTBRn_ELx.SKL
    pie : bit,          // TCR2_ELx.PIE or TCR_EL3.PIE
    pir : S1PIRType,    // PIR_ELx
    pire0 : S1PIRType, // PIRE0_EL1 or PIRE0_EL2 when HCR_EL2.E2H == '1'
    emec : bit,         // SCTLR2_EL2.EMEC or SCTLR2_EL3.EMEC
    amec : bit,         // TCR2_EL2.AMEC0 or TCR2_EL2.AMEC1 when HCR_EL2.E2H == '1'
    fng : bit,          // TCR2_EL1.FNGx or TCR2_EL2.FNGx when HCR_EL2.E2H == '1'
    fngna : bit,        // TCR2_EL1.FNGx

// A32-VMSA exclusive parameters
    t0sz : bits(3),     // TTBCR.T0SZ
    t1sz : bits(3),     // TTBCR.T1SZ
    uwxn : bit,         // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    tgX : TGx,          // TCR_ELx.TGx / Always TGx_4KB
    irgn : bits(2),     // TCR_ELx.IRGNx / TTBCR.IRGNx or HTCR.IRGN0
    orgn : bits(2),     // TCR_ELx.ORGnx / TTBCR.ORGnx or HTCR.ORGNO
    sh : bits(2),       // TCR_ELx.SHx / TTBCR.SHx or HTCR.SH0
    hpd : bit,          // TCR_ELx.HPD{x} / TTBCR2.HPDx or HTCR.HPD
    ee : bit,           // SCTLR_ELx.EE / SCTLR.EE or HSCTLR.EE
    wxn : bit,          // SCTLR_ELx.WXN / SCTLR.WXN or HSCTLR.WXN
    ntlsmd : bit,       // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
    dc : bit,           // HCR_EL2.DC / HCR.DC
    sif : bit,          // SCR_EL3.SIF / SCR.SIF
    mair : MAIRType     // MAIR_ELx / MAIR1:MAIR0 or HMAIR1:HMAIR0
};
```

Library pseudocode for shared/translation/vmsa/S2TTWParams

```
// S2TTWParams
// =====
// Register fields corresponding to stage 2 translation.

type S2TTWParams of record {
// A64-VMSA exclusive parameters
  ha : bit,           // VTCR_EL2.HA
  hd : bit,           // VTCR_EL2.HD
  sl2 : bit,          // V{S}TCR_EL2.SL2
  ds : bit,           // VTCR_EL2.DS
  dl28 : bit,         // VTCR_ELx.DL28
  sw : bit,           // VSTCR_EL2.SW
  nsw : bit,          // VTCR_EL2.NSW
  sa : bit,           // VSTCR_EL2.SA
  nsa : bit,          // VTCR_EL2.NSA
  ps : bits(3),       // VTCR_EL2.PS
  txsz : bits(6),     // V{S}TCR_EL2.T0SZ
  fwb : bit,          // HCR_EL2.FWB
  cmow : bit,         // HCRX_EL2.CMOW
  skl : bits(2),      // VTTBR_EL2.SKL
  s2pie : bit,        // VTCR_EL2.S2PIE
  s2pir : S2PIRType,  // S2PIR_EL2
  tl0 : bit,          // VTCR_EL2.TL0
  tl1 : bit,          // VTCR_EL2.TL1
  assuredonly : bit,  // VTCR_EL2.AssuredOnly
  haft : bit,         // VTCR_EL2.HAFT
  emec : bit,         // SCTLR2_EL2.EMEC
  hdbss : bit,        // VTCR_EL2.HDBSS

// A32-VMSA exclusive parameters
  s : bit,            // VTCR.S
  t0sz : bits(4),     // VTCR.T0SZ

// Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
  tgx : TGx,          // V{S}TCR_EL2.TG0 / Always TGx_4KB
  sl0 : bits(2),      // V{S}TCR_EL2.SL0 / VTCR.SL0
  irgn : bits(2),     // VTCR_EL2.IRGN0 / VTCR.IRGN0
  orgn : bits(2),     // VTCR_EL2.ORGNO / VTCR.ORGNO
  sh : bits(2),       // VTCR_EL2.SH0 / VTCR.SH0
  ee : bit,           // SCTLR_EL2.EE / HSCTLR.EE
  ptw : bit,          // HCR_EL2.PTW / HCR.PTW
  vm : bit            // HCR_EL2.VM / HCR.VM
};
```

Library pseudocode for shared/translation/vmsa/SDFTType

```
// SDFTType
// =====
// Short-descriptor format type

type SDFTType of enumeration {
  SDFTType_Table,
  SDFTType_Invalid,
  SDFTType_Supersection,
  SDFTType_Section,
  SDFTType_LargePage,
  SDFTType_SmallPage
};
```

Library pseudocode for shared/translation/vmsa/SecurityStateForRegime

```
// SecurityStateForRegime()
// =====
// Return the Security State of the given translation regime

func SecurityStateForRegime(regime : Regime) => SecurityState
begin
  case regime of
    when Regime\_EL3 => return SecurityStateAtEL\(EL3\);
    when Regime\_EL30 => return SS\_Secure; // A32 EL3 is always Secure
    when Regime\_EL2 => return SecurityStateAtEL\(EL2\);
    when Regime\_EL20 => return SecurityStateAtEL\(EL2\);
    when Regime\_EL10 => return SecurityStateAtEL\(EL1\);
  end;
end;
```

Library pseudocode for shared/translation/vmsa/StageOA

```
// StageOA()
// =====
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

func StageOA(ia : bits(64), d128 : bit, tgx : TGx, walkstate : TTWState) => FullAddress

begin
    // Output Address
    var oa : FullAddress;
    let tsize : integer = TranslationSize(d128, tgx, walkstate.level);
    let csize : integer = (if walkstate.contiguous == '1' then
        ContiguousSize(d128, tgx, walkstate.level)
    else 0);

    let ia_msb : AddressSize = (tsize + csize) as AddressSize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address[NUM\_PABITS-1:ia_msb]::ia[ia_msb-1:0];

    return oa;
end;
```

Library pseudocode for shared/translation/vmsa/TGx

```
// TGx
// ===
// Translation granules sizes

type TGx of enumeration {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```

Library pseudocode for shared/translation/vmsa/TGxGranuleBits

```
// TGxGranuleBits()
// =====
// Retrieve the address size, in bits, of a granule

func TGxGranuleBits(tgx : TGx) => AddressSize

begin
    case tgx of
        when TGx\_4KB => return 12;
        when TGx\_16KB => return 14;
        when TGx\_64KB => return 16;
    end;
end;
```

Library pseudocode for shared/translation/vmsa/TLBContext

```
// TLBContext
// =====
// Translation context compared on TLB lookups and invalidations, promoting a TLB hit on match

type TLBContext of record {
    ss : SecurityState,
    regime : Regime,
    vmid : bits(NUM\_VMIDBITS),
    asid : bits(NUM\_ASIDBITS),
    nG : bit,
    ipaspace : PASpace, // Used in stage 2 lookups & invalidations only
    includes_s1 : boolean,
    includes_s2 : boolean,
    use_vmid : boolean,
    includes_gpt : boolean,
    ia : bits(64), // Input Address
    tgx : TGx,
    cnp : bit,
    level : integer, // Assist TLBI level hints (FEAT_TTL)
    isd128 : boolean,
    xs : bit // XS attribute (FEAT_XS)
};
```


Library pseudocode for shared/translation/vmsa/TLBRecord

```
// TLBRecord
// =====
// Translation output as a TLB payload

type TLBRecord of record {
    context : TLBContext,
    walkstate : TTWState,
    blocksize : AddressSize, // Number of bits directly mapped from IA to OA
    contigsize : integer, // Number of entries log 2 marking a contiguous output range
    s1descriptor : bits(128), // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    s2descriptor : bits(128) // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
};
```

Library pseudocode for shared/translation/vmsa/TTWState

```
// TTWState
// =====
// Translation table walk state

type TTWState of record {
    istable : boolean,
    level : integer,
    baseaddress : FullAddress,
    contiguous : bit,
    s1assured : boolean, // Stage 1 Assured Translation Property
    s2assuredonly : bit, // Stage 2 AssuredOnly attribute
    disch : bit, // Stage 1 Disable Contiguous Hint
    nG : bit,
    guardedpage : bit,
    sdftype : SDFType, // AArch32 Short-descriptor format walk only
    domain : bits(4), // AArch32 Short-descriptor format walk only
    memattrs : MemoryAttributes,
    permissions : Permissions
};
```

Library pseudocode for shared/translation/vmsa/TranslationRegime

```
// TranslationRegime()
// =====
// Select the translation regime given the target EL and PE state

func TranslationRegime(el : bits(2)) => Regime
begin
    if el == EL3 then
        return if ELUsingAArch32(EL3) then Regime\_EL30 else Regime\_EL3;
    elseif el == EL2 then
        return if ELIsInHost(EL2) then Regime\_EL20 else Regime\_EL2;
    elseif el == EL1 then
        return Regime\_EL10;
    elseif el == EL0 then
        if CurrentSecurityState() == SS\_Secure && ELUsingAArch32(EL3) then
            return Regime\_EL30;
        elseif ELIsInHost(EL0) then
            return Regime\_EL20;
        else
            return Regime\_EL10;
        end;
    else
        unreachable;
    end;
end;
```

Library pseudocode for shared/translation/vmsa/TranslationSize

```
// TranslationSize()
// =====
// Compute the number of bits directly mapped from the input address
// to the output address

func TranslationSize(d128 : bit, tgx : TGx, level : integer) => AddressSize
begin
    let granulebits : AddressSize = TGxGranuleBits(tgx);
    let descsize2 : integer{} = if d128 == '1' then 4 else 3;
    let blockbits : integer = (FINAL\_LEVEL - level) * (granulebits - descsize2);

    return (granulebits + blockbits) as AddressSize;
end;
```

Library pseudocode for shared/translation/vmsa/UseASID

```
// UseASID()
// =====
// Determine whether the translation context for the access requires ASID or is a global entry

func UseASID(accesscontext : TLBContext) => boolean
begin
    return HasUnprivileged(accesscontext.regime);
end;
```

Library pseudocode for shared/translation/vmsa/UseVMID

```
// UseVMID()
// =====
// Determine whether the translation context for the access requires VMID to match a TLB entry

func UseVMID(regime : Regime) => boolean
begin
    return regime == Regime\_EL10 && EL2Enabled();
end;
```

2026-03_rel 2026-03-26 20:48:11

Copyright © 2010-2026 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.